ISSN 0265-2919

80p 35

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO

An ORBIS Publication

IR £1 Aus $1.95 NZ $2.25 SA R1.95 Sing $4.50 USA & Can $1.95

CONTENTS

Next Week

■ The Plus 4 is the latest home micro from Commodore with a new look, new ROMs and a new version of BASIC. Our review measures it against the competition.

■ Having examined LOGO sprites on the Commodore 64, we investigate the Atari version.

■ A robot that can sense its surroundings and move purposively amongst them is approaching intelligence. Our robotics series considers intelligent movement.

QUIZ

1) What happens to a 'grandfather file'?
2) What are decoders?
3) What do transducers do?
4) Where would you find the system variable 'PAGE'?

Answers To Last Week's Quiz

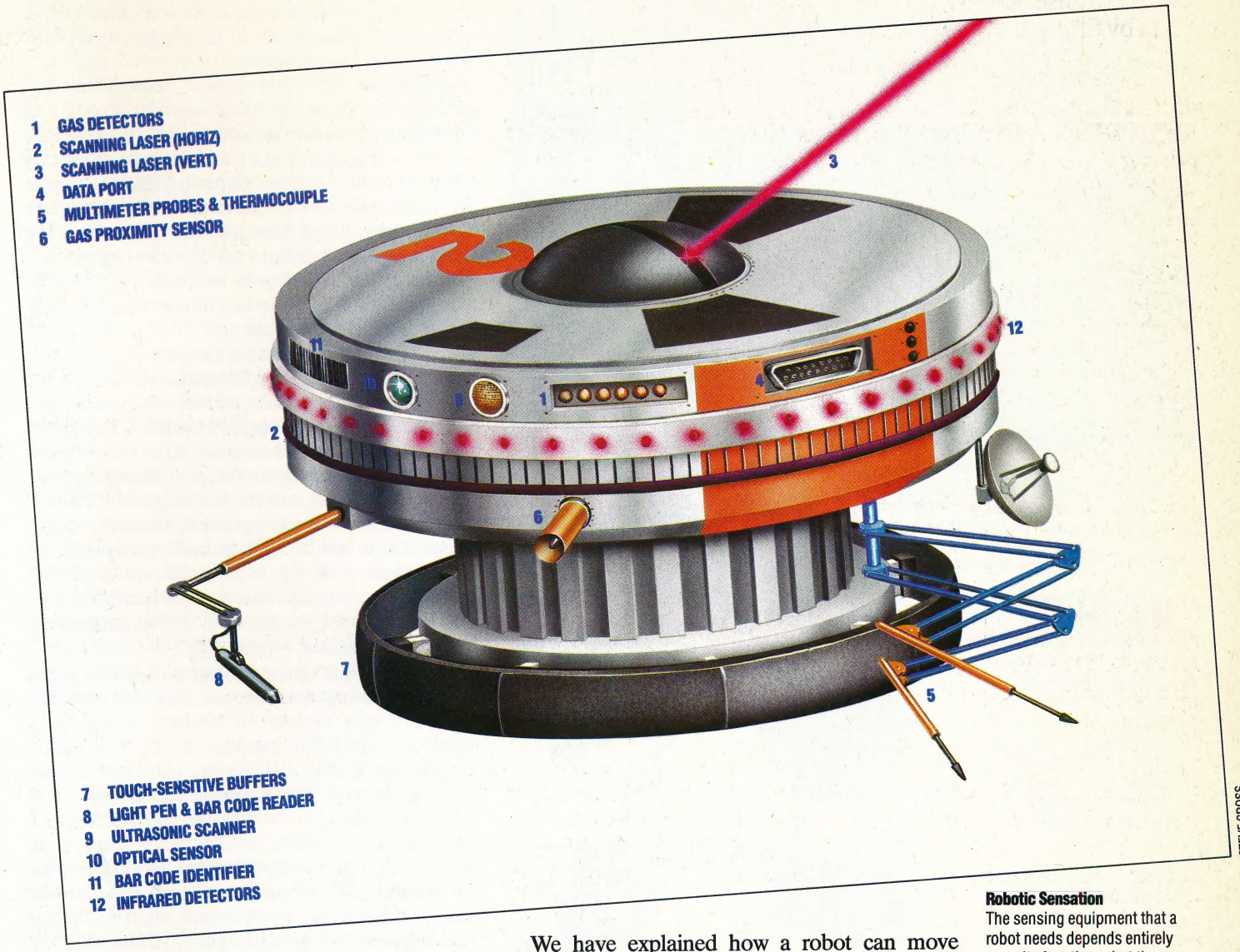
1) The five keywords on the MSX power-up display are the commands currently programmed into the function keys.
2) A UART is a Universal Asynchronous Receiver/Transmitter. It converts serial data into parallel and vice versa.
3) Cartesian co-ordinates are named after the 17th century philosopher, René Descartes, who founded analytic geometry.
4) The adventure game 'Star Trek', based on the American television series, was the model for Atari's 'Star Raiders'.

Editor Mike Wesley; **Art Director** David Whelan; **Technical Editor** Brian Morris; **Production Editor** Catherine Cardwell; **Art Editor** Claudia Zeff; **Chief Sub Editor** Robert Pickering; **Designer** Julian Dorr; **Art Assistant** Liz Dixon; **Editorial Assistant** Stephen Malone; **Sub Editor** Steve Mann; **Researcher** Melanie Davis; **Staff Writer** Steve Colwill; **Contributors** Geoff Bains, Harvey Mellor, Mike Curtis, Steve Colwill, Chris Naylor, Tony Harrington, Jim Lennox, Steve Malone, Ted Ball; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd: Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Maurice Geller; **Production Controller** Peter Taylor-Medhurst; **Circulation Director** David Breed; **Marketing Director** Michael Joyce; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; **© APSIF Copenhagen 1984; © Orbis Publishing Ltd 1984: Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Artisan Press Ltd, Leicester**

HOW TO OBTAIN ISSUES AND BINDERS FOR THE HOME COMPUTER ADVANCED COURSE – Issues can be obtained by placing an order with your newsagent or direct from our subscription department. If you have any difficulty obtaining any back issues from your newsagent, please write to us stating the issue(s) required and enclosing a cheque for the cover price of the issue(s). **AUSTRALIA** – please write to: Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G, Melbourne, Victoria 3001. **MALTA, NEW ZEALAND & SOUTH AFRICA** – Back numbers are available at cover price from your newsagent. In case of difficulty, write to the address given for binders. **UK/EIRE** – Price: 80p/IR£1. Subscription: 6 months: £23.92. 1 Year: £47.84. Binder: please send £3.95 per binder, or take advantage of our special offer in early issues. **EUROPE** – Price: 80p. Subscription: 6 months air: £37.96. Surface: £31.46. 1 year air: £75.92. Surface: £62.92. Binder: £5.00. Airmail: £5.00. **MALTA** – Obtain binders from your newsagent or Miller (Malta) Ltd, MA Vassalli Street, Valetta, Malta. Price: £3.95. **MIDDLE EAST** – Price: 80p. Subscription: 6 months air: £43.94. Surface: £31.46. 1 year air: £87.88. Surface: £62.92. Binder: £5.00. Airmail: £8.31. **AMERICAS/ASIA/AFRICA** – Price: US/CAN$1.95/80p. Subscription: 6 months air: £51.74. Surface: £31.46. 1 year air: £103.48. Surface: £62.92. Binder: £5.00. Airmail: £9.44. **SOUTH AFRICA** – Price: SA R1.95. Obtain binders from any branch of Central News Agency or Intermag, PO Box 57394, Springfield 2137. **SINGAPORE** – Price: Sing $4.50. Obtain binders from MPH Distributors, 601 Sims Drive, 03-07-21, Singapore 1438. **AUSTRALASIA/FAR EAST** – Price: 80p. Subscription: 6 months air: £55.38. Surface: £31.46. 1 year air £110.76. Surface: £62.92. Binder: £5.00. Airmail: £9.84. **AUSTRALIA** – Price: Aus$1.95. Obtain binders from First Post Pty Ltd, 23 Chandos Street, St Leonards, NSW 2065. **NEW ZEALAND** – Price: NZS2.25. Obtain binders from your newsagent or Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington.
ADDRESS FOR BINDERS AND BACK ISSUES – Orbis Publishing Limited, Orbis House, Bedfordbury, London WC2 4BT. Telephone 01-379 6711. Cheques/postal orders should be made payable to Orbis Publishing Limited. Binder prices include postage and packing and prices are in stirling. Back issues are sold at the cover price, and we do not charge carriage in the UK.
NOTE – Binders and back issues are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK and Australian markets only. Binders and Issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.
ADDRESS FOR SUBSCRIPTIONS – Orbis Publishing Limited, Hurst Farm, Baydon Road, Lambourn Woodlands, Newbury Berks, RG16 7TW. Telephone: 0488-72666. All cheques/postal orders should be made payable to Orbis Publishing Limited. Postage and packaging is included in subscription rates, and prices are given in sterling.

COVER PHOTOGRAPHY BY IAN McKINNELL

SENSE & SENSIBILITY



STEVE CROSS

Robotic Sensation
The sensing equipment that a robot needs depends entirely upon its functions, but the more general-purpose the robot, the more sensors it is likely to need. The robot illustrated has examples of most of the possible sensors available, though it is unlikely that any single robot would carry such a range

Our series of articles has now considered methods of controlling robot movement and the design of robotic 'arms' and 'hands'. We now turn our attention to the ways in which robots can 'sense' what is happening in the world around them.

The human sensory apparatus is something that we take very much for granted, but a person lacking all senses would be totally helpless. Without a sense of sight, you would bump into objects as you tried to walk; lacking a sense of touch you would not even know that you *had* bumped into anything; total deafness would mean that you could not even receive a warning that you were about to walk into an object. In fact, you wouldn't be able to walk at all, because internal senses are required to inform your brain of the way in which your body is moving.

We have explained how a robot can move around, but we must also provide it with a sensory system before it can act independently. It is tempting to try to design a robot that possesses *all* human senses — it could then perceive the world in much the same way as we do. As yet, however, this is impossible. The subjects of robot vision and speech understanding are so complex that we will deal with them at length in a later article. Here we will concentrate on simple approximations of sight and hearing that are far below the level of complexity possessed by humans.

A robot can be made to 'see' things quite easily by providing it with a light sensor — usually a photoelectric cell — which produces a voltage that varies with the amount of light falling on it. This is a very crude vision sensor, but it can be used to good effect. For example, a robot can be made to 'home in' on a bright light in much the same way as it can follow a line (see page 641). This might be used to allow the robot to return to a power point

for recharging when its batteries run down. (Note that this will require the robot to possess an internal sensor for monitoring the state of its batteries so that it 'knows' when they are low.)

This simple photoelectric cell can allow a robot to perform a number of tasks. A robot working on an assembly line would be able to check if a component was present by detecting the change in brightness that results from the object's absence; this task can be made easier by ensuring that the lighting is arranged so that such a difference is accentuated. The robot could detect colour changes if three photoelectric cells were incorporated, each responding to light of different colours — red, green and blue would cover the visible spectrum. Such a robot could be programmed to pick out red bricks from a pile containing bricks of many different colours. This gives the impression of 'intelligent' behaviour from a very simple sensor.

Providing the robot with a microphone will allow it to 'hear' acoustic signals. It will not 'understand' what it is hearing, but this need not matter — by repeating a set of commands several times, the robot can build up a 'template' of sound for each command that will enable it to match new commands against those it has already heard. The number of commands to which it can respond will be limited, but we could tell it to 'go forward', 'stop', 'turn left', and so on, and it would be able to follow these instructions.

A robot can also have a simple sense of touch. Microswitches can be incorporated into the robot's design so that they make an electrical connection whenever pressure is applied to them. These lack the sophistication of the human sense of touch, but they can still be very useful. For example, touch sensors mounted around the edge of a mobile robot can allow it to respond intelligently to any obstacles: it will be able to back away from the obstruction and try a different route. Touch sensors incorporated into a robot hand will let it 'know' when it has something within its grasp so that it can respond accordingly.

Smoke or gas detectors can be used to give a robot something of a sense of smell. Gas detectors normally use a sensory element (such as a platinum wire) that responds to the presence of certain gases, thus altering the electrical current flowing through the element. Smoke detectors have two chambers — one enclosed, acting as a reference or 'control', and the other open. Both chambers contain ionised helium and the number of charged particles in the open chamber varies when smoke is present. A detector that counts the number of charged particles in each chamber will register a difference between the two if smoke is present.

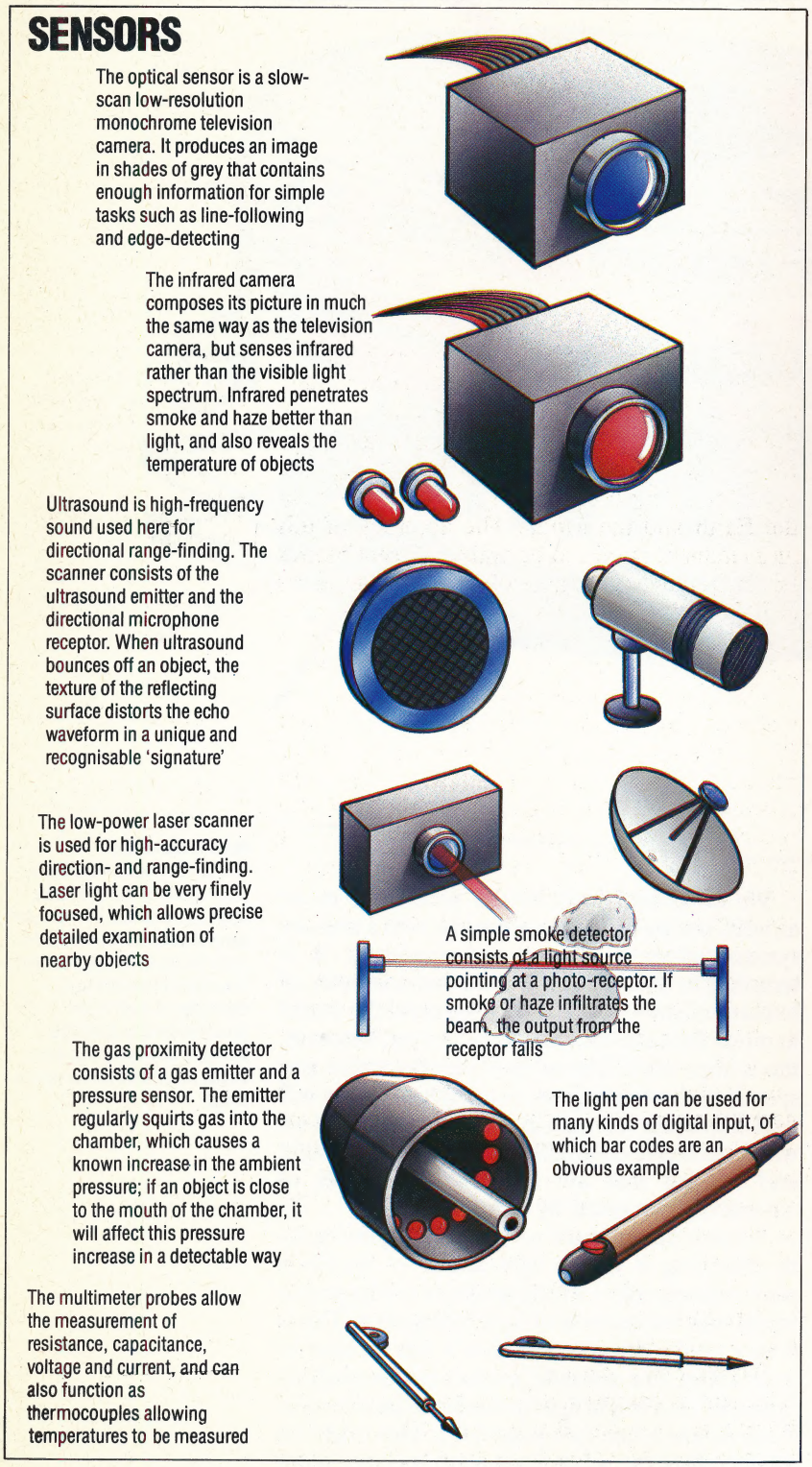
As yet, there appears to be no way in which a robot can be given a sense of taste. However, using the methods suggested above, we at least have a robot that can see, hear, feel and smell well enough to detect a fire in a building, rush towards the flames, avoid obstacles in its way and, if it happens to have a fire extinguisher in its end effector, spray the fire with foam.

POTENTIAL GAIN

But in limiting a robot to the type of senses possessed by humans, we lose much of its potential. There is no reason for the robot to be restricted to detecting things in the ways that we detect them. A better approach might be to consider what senses can be given to a robot and to decide if there is any practical use for them.

A good example of this concerns robot arms.

SENSORS



KEVIN JONES

Let's assume that we want a robot to pick up an object from one location and then place it somewhere else. One way of doing this is to fix end stops around the arm so that it can travel a set maximum distance in any given direction. The arm would then swing around until it came to the end stops, at which point (if everything was positioned correctly) the hand would be directly above the object to be grasped. After picking up the object, the arm could swing in the opposite direction until another set of end stops would let it know that the object should be released. This is a simple example, and it is one that is becoming less common, but it demonstrates that robots can use senses that are not possessed by humans.

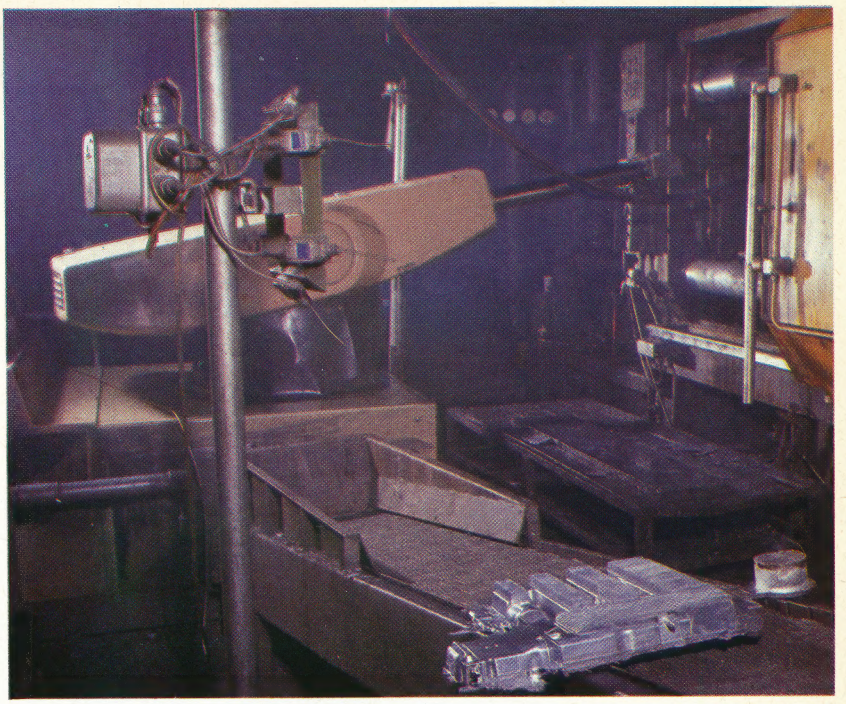
Perhaps a better example is a robot's use of 'vision'. Humans see only visible light — a large part of the electromagnetic spectrum is invisible to the human eye — but there is no reason why a robot should be so confined. Infrared detectors may be fitted in place of photoelectric cells; these will allow the amount of heat generated by an object to be measured. Industrial robots can make use of these detectors to move away from any dangerously hot objects, for example. But a robot could also detect the warmth of a human body — so your personal robot could be programmed to come running to greet you as you walk through your front door! Robots can also be made to detect magnetic fields. This has already been discussed in connection with robots that follow a track laid in the ground, but this facility would also be useful in applications in which a robot must differentiate between magnetic and non-magnetic materials.

Proximity sensors have no direct human equivalent — these are merely devices that can detect when an object is near them. Humans use a combination of sight and touch for this purpose, but a simple proximity sensor is just as suitable for robot use. Such sensors work in a number of different ways. One type uses an air jet squirted through a nozzle; any object in the jet's path will deflect the air back towards the nozzle. This creates a back pressure that may be detected by a pressure transducer, thus warning the robot that something is near. Another type depends on the fact that an electrical circuit with a capacitance will change its behaviour if it approaches another object. An electrical 'leak' between the capacitor and the object (which will have a capacitance of its own) will inform the robot that an object is nearby.

TRANSDUCERS

There are also ultrasonic sensors that work by emitting an ultrasonic signal and then picking up the echo from the nearby object. The time delay between the signal and the echo gives an exact measurement of the distance from the object. This is similar to the method used by bats for navigation, and the principle is also used in some auto-focusing cameras.

More sophisticated still are laser sensors. These direct a laser beam onto an object, which then reflects the laser light back to the sensor. By comparing the two beams it is possible to determine the object's distance with astonishing accuracy. This technique can be used over great distances. During the first manned lunar landing, a reflector was placed on the moon so that a laser sensor could measure the exact distance between the Earth and the Moon. The accuracy of this measurement is said to be within 15 centimetres (six inches) over a distance of 384,400 kilometres (250,000 miles)!



No Sense, No Feeling
This industrial robot arm is cleaning castings straight from the mould when they are still too hot to be touched by human hands. The robot is impervious to heat, of course, and therefore processes the work more quickly

Force sensors are a means of obtaining tactile information by more sophisticated means than mechanical microswitches. These work by measuring the change in the electrical properties of a piezo-electric crystal when it comes under pressure, or by calculating the change in conductivity of carbon graphite granules under pressure (using a technique identical to that employed in the carbon microphone). Alternatively, strain gauges can be used to measure large forces by detecting changes in the electrical resistance of a wire as it is stretched.

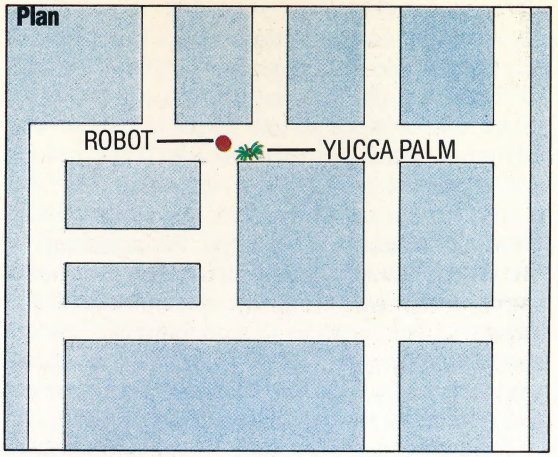
These robot sensors come under the heading of 'transducers', as they take a measurement in one form (such as light, sound or pressure) and convert ('transduce') it into another form that in some way represents the original measurement. On a computer-controlled robot, the transducers almost invariably convert the measurement into an electrical signal that may be binary (i.e. the signal is either present or absent) or analogue (the signal varies as the original measurement changes). In the latter case, the electrical signal must be converted into a form the computer can understand by using an analogue-to-digital (A/D) converter.

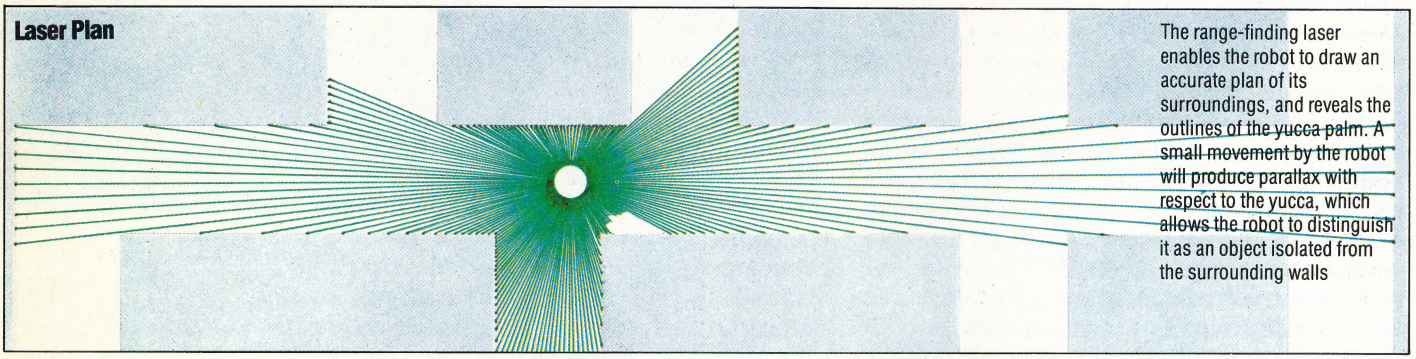
It is fair to say that a robot's senses are not, at the moment, as comprehensive or as effective as the human equivalents. But the robot has more of them — and they are getting better all the time.

Sense From The Sensors

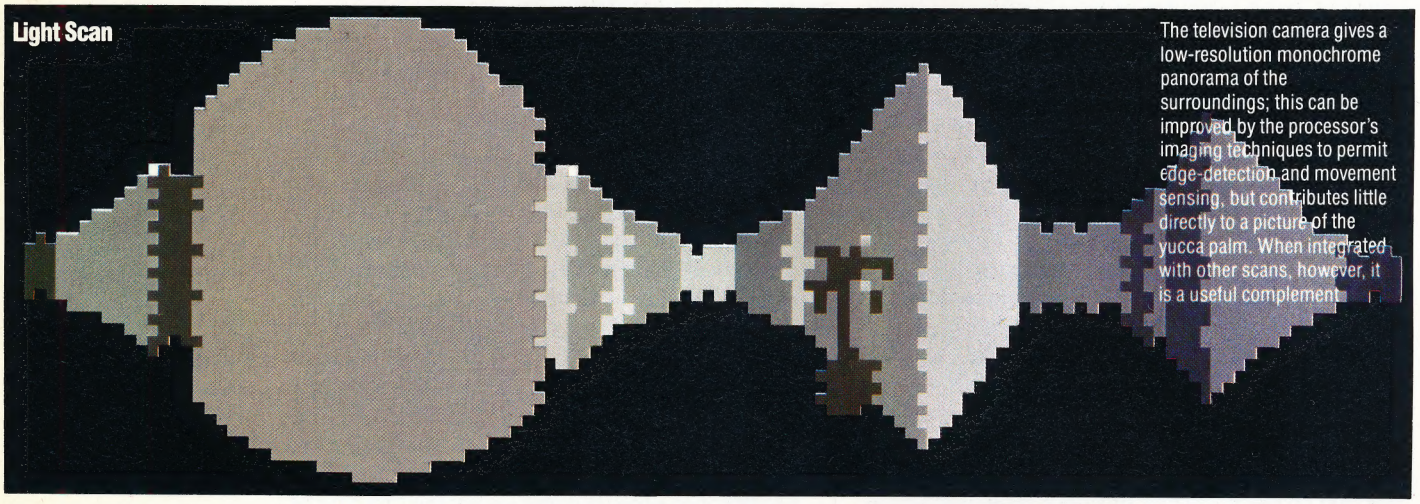
Making sense of the outside world is the robot's greatest problem, the more so as the range and complexity of its sensing equipment increases. No single sensor will give a completely informative picture, and some may seem to contradict one another. The extent to which the robot can integrate and compare the input from its various sensors is the measure of its external 'consciousness'.

The plan shows that, in this example, the robot is in a corridor whose walls are painted flat white; there is only one light source, so the illumination of a wall depends upon its orientation. Near the robot is a yucca palm

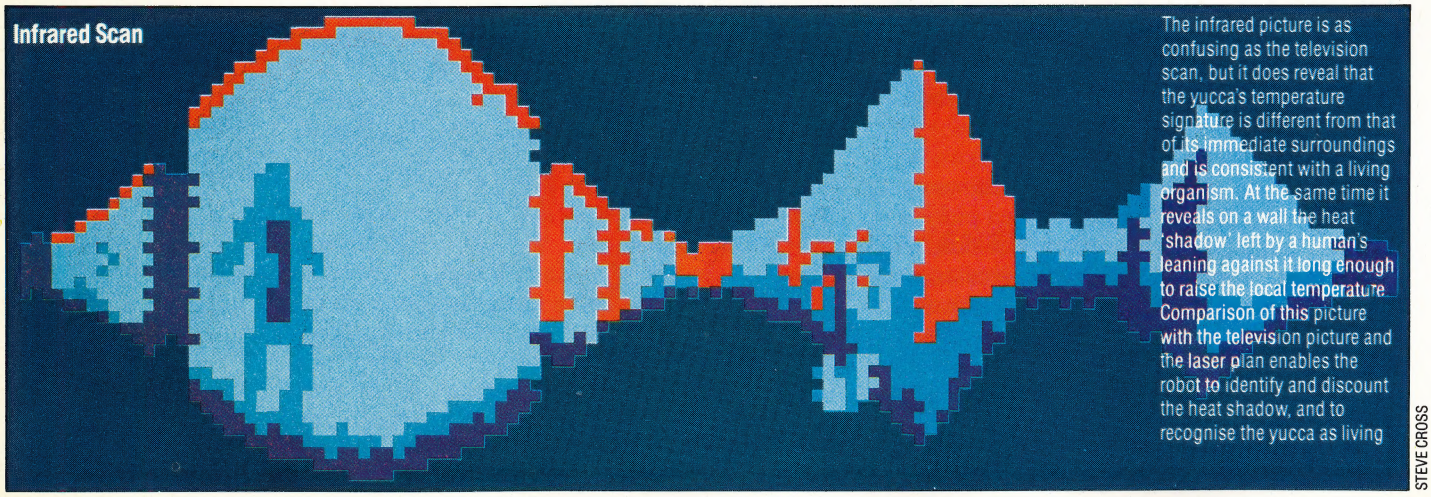




The range-finding laser enables the robot to draw an accurate plan of its surroundings, and reveals the outlines of the yucca palm. A small movement by the robot will produce parallax with respect to the yucca, which allows the robot to distinguish it as an object isolated from the surrounding walls



The television camera gives a low-resolution monochrome panorama of the surroundings; this can be improved by the processor's imaging techniques to permit edge-detection and movement sensing, but contributes little directly to a picture of the yucca palm. When integrated with other scans, however, it is a useful complement



The infrared picture is as confusing as the television scan, but it does reveal that the yucca's temperature signature is different from that of its immediate surroundings and is consistent with a living organism. At the same time it reveals on a wall the heat 'shadow' left by a human's leaning against it long enough to raise the local temperature. Comparison of this picture with the television picture and the laser plan enables the robot to identify and discount the heat shadow, and to recognise the yucca as living

STEVE CROSS

DISPLAY COUNTER

In this instalment of Workshop we add two seven-segment displays to our user port system that will enable us to display the contents of the user port data continuously in hexadecimal.

In order to display hexadecimal digits, four bits are required (four bits give us 16 permutations of zeros and ones). Thus, any eight-bit number can be represented using two hexadecimal digits: one for the lower four bits and another for the upper four. Although each display is made up of seven LED segments, the various combinations of segments can be 'driven' by four input lines if decoder logic is incorporated into the circuit.

Decoders are circuits that translate instructions from the computer to its peripherals into electrical signals, and vice versa. In our series on logic we built our own decoder circuit (see page 146), but for this exercise we can buy an off- the-shelf logic circuit. This is chip 7447 in the parts list.

The decoder for each display accepts four input lines from the user port and, via a sequence of logic gates, provides seven outputs. The logic circuit has been designed in such a way that if, say, the four input lines were 0111, then the appropriate bars would be lit to display the figure 7 (0111 in binary is equivalent to 7 in hexademical). The truth table for this is shown in the margin.

Hexadecimal digits greater than nine are usually represented by the first six letters of the alphabet — A to F. You will notice that the decoder chip that we are using has rather strange patterns to represent these digits. It is probable that these patterns can be generated using the logic circuits required for the digits zero to nine. More decoding logic would be needed to display the last six hex digits in the more usual alphabetic way, so by leaving the extra logic out and using different symbols for these digits, the number of logic gates in the decoder is reduced, thus bringing down the cost of manufacturing the chip.

Once the display circuit has been built we can display continuously the contents of the user port data register in hex using the eight data lines provided. There are sufficient lines available to drive the two displays simultaneously, but in many display applications this is not the case and several seven-segment displays have to share the same data lines. So that each display can show different information at the same time, a technique called 'multiplexing' is used. Essentially, the data lines from the display decoder are flipped from one display to the next, the data present on the lines also being changed appropriately. If this is done fast enough all the displays multiplexed in this way will appear to glow continuously, each displaying the data that was present at the time when it is momentarily connected to the data lines.

Parts List

No	Item	Maplin No
14	330 ohm 0.4 watt resistors	M330R
2	7447 BCD to 7-segment decoder	QX55K
1	Common anode double digit display	BY66W
2	16-pin DIL chip socket	BL19V
1	12-way minicon right-angle socket	YW30H
1	10-way minicon right-angle plug	YW19V
	8-way ribbon cable*	
	7-way ribbon cable*	
	Tinned bare wire*	
1	50 hole x 36 strip veroboard	FL09K
1	116 x 61 x 36mm plastic box	LH60Q

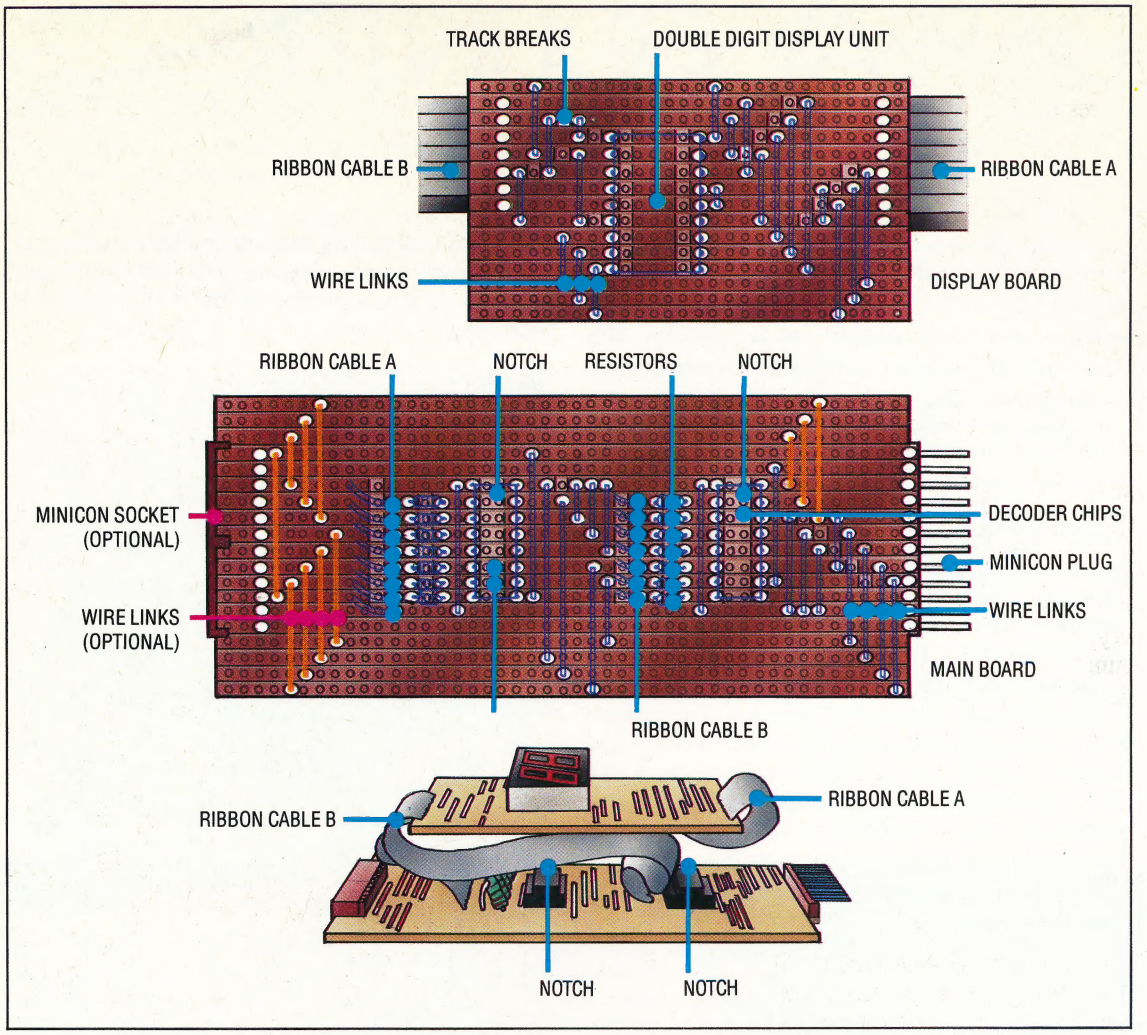
*You may have these parts left over from previous projects. The 12-way socket is necessary only if you want to extend the system bus. This, and several wire links on the circuit board may be omitted

We can demonstrate the principle of multiplexing using the two seven-segment displays that we are building. Because the display decoder represents decimal 15 by a blank, we can use this number to blank out one display while lighting the other. The following program, when run, asks for a digit to be displayed and then appears to show the digit on both displays simultaneously. However, a routine is included that inserts a delay to slow down the oscillation between the two displays. The delay is inserted while the Space bar is depressed. We can see, on running the program and depressing the Space bar, that the digit does in fact flip backwards and forwards between the two displays. When the Space bar is again released, the delay is removed and the flipping action is faster, making the digit seem to appear simultaneously on each display.

```
 10 REM BBC MULTIPLEXING
 20 DDR=&FE62:DATREG=&FE60
 30 ?DDR=255
 40 left_blank=15*16
 50 right_blank=15
 55 :
 60 REPEAT
 70 INPUT"DATA TO BE MULTIPLEXED";data
 80 ?DATREG=data+left_blank
 90 PROCslower
100 ?DATREG=data*16+right_blank
110 PROCslower
120 GOTO80
130 END
140 :
150 DEF PROCslower
155 REM IS SPACE BAR PRESSED ?
```

Laying It Out

Cut the veroboard to the two sizes required (19 tracks of 46 holes; 15 tracks of 28 holes). Cut the track breaks first on both boards. Solder the two chip sockets in place first, then the wire links and the resistors. If the bus extension socket is not required, then omit the red-coloured links in the illustration. Fit the minicon plug and (optional) socket to the main board, and solder the display unit in place — dots towards the socket end of the board. Solder the connection ribbon cables in place, so that they go straight from board to board without twisting. Now plug in the chips — make sure they are oriented as shown



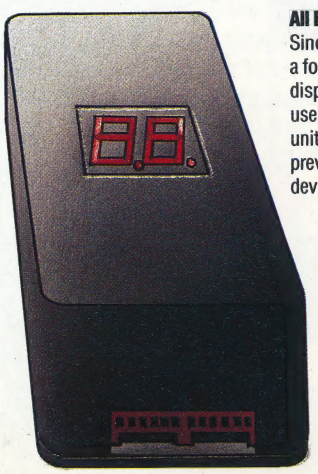

```
160 IF INKEY(-99)=-1 THEN PROCdelay
170 ENDPROC
180 :
190 DEF PROCdelay
200 FOR I =1 TO 500:NEXT
210 ENDPROC

10 REM CBM 64 MULTIPLEXING
30 DDR=56579:DATREG=56577
40 POKEDDR,255
50 LB=15*16:RB=15
60 INPUT"DATA TO BE MULTIPLEXED";DT
70 POKEDATREG,DT+LB
80 GOSUB1000:REM SLOWER
90 POKE DATREG,DT*16+RB
100 GOSUB1000:REM SLOWER
110 GOTO70
120 :
1000 REM SLOWER S/R
1010 GETA$
1020 IFA$="" "THENGOSUB2000:REM DELAY
1030 RETURN
1999 :
2000 REM DELAY S/R
2010 FORI=1 TO 250:NEXT
2020 RETURN
```

A simple application is to use the twin seven-segment displays as a hexadecimal counter. This displays a count of the number of pulses input to the user port from a simple make-or-break switch connected to one port line. On first glance this seems a trivial task until you realise that all eight user port lines are needed for the display, leaving none for input. If we specify one of the lines for input, say line 0, then the computer's I/O system will always hold this line high, no matter what number may be present in the data register. If 128 (10000000 in binary) were to be placed in the data register, then this would instantly be changed to 129 (10000001) because line 0 is held high for input. This would obviously give incorrect count values on the displays. The solution lies in using a technique similar to multiplexing. If we set line 0 to accept input for a short time only and set all lines to output for a longer time, then the displays will appear to glow continuously with the correct counter value, with only a flicker of the incorrect value caused by setting line 0 for input.

```
10 REM BBC COUNTER
20 DDR=&FE62:DATREG=&FE60
50 count=0
55 :
60 REPEAT
70 PROCinput
72 PROCadd
73 FORI=1TO40
75 PROCdisplay
77 NEXT I
80 UNTIL count>255
90 END
999 :
1000 DEF PROCadd
1010 IF flag =1 THEN count=count+1
1050 ENDPROC
```

```
1499 :
1500 DEF PROCinput
1510 ?DDR=254
1515 flag=0
1520 IF(?DATREG AND 1)=0 THEN flag=1
1525 REPEAT UNTIL (?DATREG AND 1)=1
1530 ENDPROC
1999 :
2000 DEF PROCdisplay
2010 ?DDR=255
2030 ?DATREG=count
2040 ENDPROC

10 REM CBM 64 COUNTER
30 DDR=56579:DATREG=56577
40 CC=0:REM INIT COUNT
50 :
60 GOSUB1000:REM INPUT
70 GOSUB2000:REM ADD
80 FOR I=1TO20
90 GOSUB3000:REM DISPLAY
100 NEXT I
110 IF CC<255 THEN60
120 END
999 :
1000 REM INPUT S/R
1010 POKEDDR,254
1020 FL=0
1030 IF(PEEK(DATREG)AND 1)=0 THEN FL=1
1040 IF(PEEK(DATREG)AND 1)<>1 THEN 1040
1050 RETURN
1999 :
2000 REM ADD S/R
2010 IF FL=1 THEN CC=CC+1
2020 RETURN
2999 :
3000 REM DISPLAY S/R
3010 POKEDDR,255
3020 POKEDATREG,CC
3030 RETURN
```

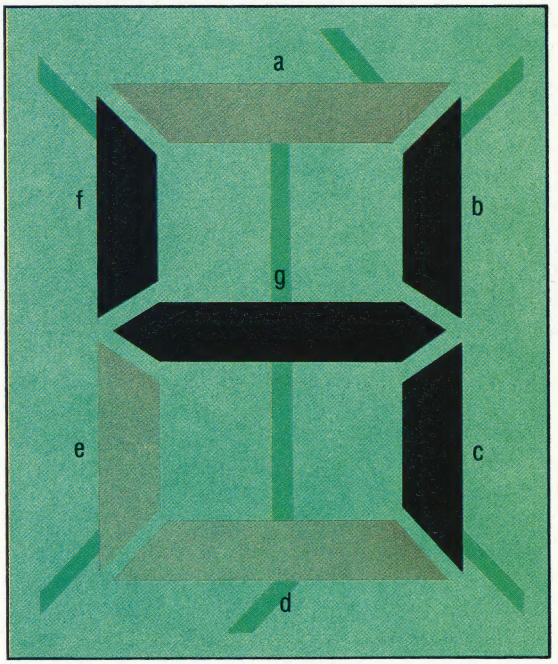
In each cycle of the program, a FOR... NEXT loop is used to repeat many times the routine where all lines are set to output, for each routine where line 0 is set for input. For the Commodore 64 version, 20 display routines are executed for each input routine. This ratio is increased to 40 for the BBC version due to the increased speed of execution of the BBC Micro. With these ratios, a flicker is still detectable, but if the ratio is increased still further to produce a smoother display, then it is possible that the amount of time spent looking for an input is reduced so much that inputs may be missed altogether. In 'real-time' electronics such compromises between conflicting demands of parts of a system have to be made.

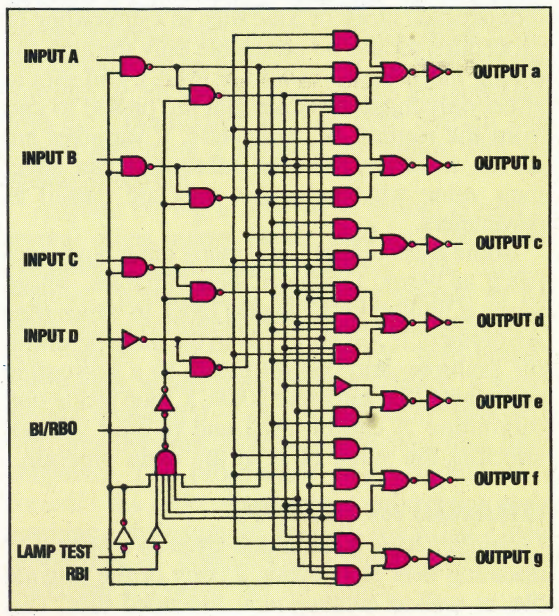


All Boxed Up
Since each digital display needs a four-bit input code, two displays can be driven from the user port. They are boxed in one unit compatible with our previously-built interfaces and devices

Decimal	Binary D3 D2 D1 D0	Output a b c d e f g	Display
0	0 0 0 0	0000001	0
1	0 0 0 1	1001111	1
2	0 0 1 0	0010010	2
3	0 0 1 0	0000110	3
4	0 1 0 0	1001100	4
5	0 1 0 1	0100100	5
6	0 1 1 0	1100000	6
7	0 1 1 1	0001111	7
8	1 0 0 0	0000000	8
9	1 0 0 1	0001100	9
10	1 0 1 0	1110010	c
11	1 0 1 1	1100110	ɔ
12	1 1 0 0	1011100	u
13	1 1 0 1	0110100	ɔ
14	1 1 1 0	1110000	t
15	1 1 1 1	1111111	

Figure It Out
The input to the digital display from the user port is a four-bit binary number. Associated with each of the numbers 0000 to 1111 is a unique seven-bit number, each bit of which signals the state of one of the seven segments of the display. In this display code a zero bit means that the corresponding segment is to be lit, and a one indicates an unlit segment





7447A BCD To 7-Segment Decoder/Driver
The internal circuitry of the chip shows the essential simplicity of its logic — the four-bit input is decoded into the seven segment outputs by the logic gates. The lamp test input switches all segments on simultaneously to test the chip

GATES

The electronic circuitry that comprises most of a computer's working parts consists of thousands of different kinds of switches: information is represented in the machine by patterns of electrical currents, and its processing is effected by switching it through the various routes that these switches, or *gates*, provide. Gates are so called because their function is to allow or deny the passage of information according to the properties of the information and the nature of the gate.

The elementary gates are the AND and OR gates, corresponding to the logical operations of Boolean algebra (see page 32). All logical expressions can be reduced to expressions of these operators, so all the processing functions of the computer can be built using only these gates. The practicalities of integrated circuit construction, however, make NAND and NOR (AND and OR gates with an inverter on the output) cheaper and more convenient to use in large quantities.

GLOBALITY

In a comparatively simple programming language such as BASIC, the variables used in one part of a program are usually available in all other parts of the program, so if X is initialised as 134, say, at the start of the program, then any subsequent reference to X will be to that variable containing the value 134. Such variables are said to be *global* in scope. This is often a convenient feature, and is so commonplace in BASIC that alternatives are rarely considered. However, it can present problems, particularly in large programs, and especially those written by more than one programmer, or when library routines are merged with existing programs. Using a variable in one part of a program, then inadvertently re-using it elsewhere for a different purpose is a very common error, especially when the dialect insists on single-character loop-counter variables. The real shortcoming of global variables is the threat they pose to program structure. A logically independent block of code such as a subroutine or procedure should be accessible only through the calling mechanism (GOSUB or PROC), and should affect only those variables that are passed to and from the routine as parameters; if variables are global, however, then the subroutine can affect their value whether or not they are passed as parameters.

Some BASIC dialects, and many other languages (such as PASCAL), support *local* variables, whose scope is limited to the logical block in which they are defined. If X is set at 134 in the main program, for example, and control passes to a subroutine containing the statement LOCAL X, then inside that subroutine X can be used and re-used without affecting the value of X in the main program.

GRANDFATHERING

Grandfathering is the name given to a system of updating files that retains a copy of the original file, as well as creating new, amended files. This system ensures the safe storage of information, as if a file is accidentally destroyed, there is always a copy in existence. Once a file is amended it is referred to as the *father* file, and the file from which it was created the *grandfather* file. Any subsequent update of the father file is known as the *son* file. As the file is updated, the latest version is always the son file and the most out-of-date the grandfather file. Only three *generations* of files are stored at once, so as new files are created, the current grandfather file is deleted.

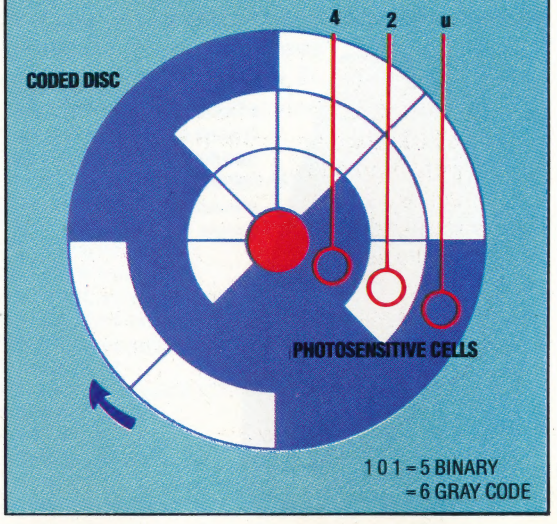
GRAY CODE

In computerised control applications, positional data written on a moving object must often be read by a mechanical reader. The accuracy of the reading is subject to errors in the timing of the read cycle, and it often happens that a small error causes the sensors to point to the gaps between the data rather than to the data itself. It is then a matter of chance which of the two numbers is read by the sensor. The *Gray code* is a way of encoding binary data to minimise the effect of such errors. The principle of the code is that only one bit of a number changes with every successive increment of the number, and the bit that does change should be as far to the right in the number as possible. Compare these numbers in binary and Gray code:

Decimal	Binary	Gray Code
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111

If the mechanical reader is positioned between two consecutive Gray code numbers and reads the wrong one, then the error is confined to one bit, since there is only one bit that changes.

Gray Disc
The disc is attached to a rotating shaft to enable the photocells to read the shaft's position. The three parts of each segment are coloured to represent the numbers zero to seven in Gray code and are read by the photocells positioned above the disc. Each segment of the disc differs from its neighbour in only one respect — the basis of the Gray code



L72 DIXON

SHOPPING AROUND

As new and improved machines are appearing on the market all the time, many people find themselves replacing or upgrading their computers every couple of years. Here we present some advice on buying a home computer, and give some indication of the strengths and weaknesses of the more popular models.

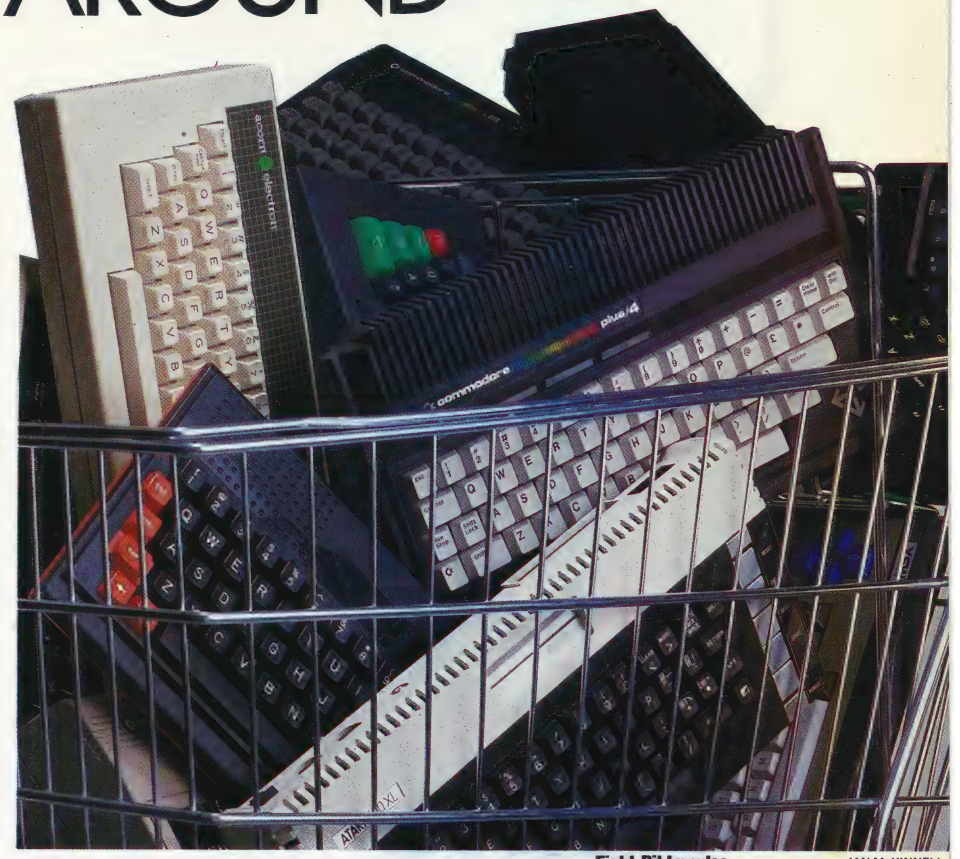
When buying a new machine, it is important to consider exactly what your needs are — do you want a computer that can be expanded by the addition of peripherals, extra memory, etc., or can you afford to treat it as a disposable item, to be sold off when something better comes along?

Most of the newer models boast more features than their older rivals, including bigger memories, better dialects of BASIC, higher resolution graphics and built-in software. But the older machines, particularly those that have sold in large quantities, have one major advantage — software availability. Many buyers of newer computers will have to wait for months before a large range of software is available — and in some cases, this software will never appear. A good example of this is the Oric Atmos. This improved version of the Oric-1 has been on sale for months, yet software writers have been reluctant to produce material for it. As a result, the sales figures for this machine have declined drastically.

The three micros that are best served by software houses are the Sinclair Spectrum, the Commodore 64 and the BBC Micro. The Spectrum, in particular, is a prime example of the way in which creative software writing can overcome a machine's inbuilt limitations: some of the programs for this micro can compare very favourably with those produced for considerably more sophisticated machines. However, it is unlikely that any of these three machines would sell well if they were launched today: the Spectrum has an extremely poor keyboard, the Commodore 64's BASIC lacks the commands to make the most of the computer's potential, and the BBC Micro has a small memory and is decidedly overpriced by today's standards.

Most of the newer micros have more impressive specifications, but lack the depth and breadth of software. Anyone buying one of these newer machines is gambling that it will gain popularity and hence convince software developers to generate programs for it.

The major trend with the new home computers is to provide more for your money. High quality keyboards, larger user memories (64 Kbytes or more) and good graphics are now standard. The quality of the BASIC interpreter has been improved considerably in machines like the Commodore Plus/4, Commodore 16, Sinclair QL and the MSX micros. The Amstrad machine even includes a monochrome or colour monitor in its price.



Eight-Bit Impulse IAN McKINNELL
Shopping for computers should be just like shopping for clothes, but somehow the weight of technical information and the range of choice combine to make it more like a lucky dip. A cool matching of your needs to the machines' capabilities is an essential precursor to visiting the computer store; try to decide in advance what you're going to buy, and then let the 'feel' of the chosen machine be the last — not the first — deciding factor

Another interesting new trend is the provision of 'bundled', or free, software. The Sinclair QL is supplied with four such programs: a word processor, spreadsheet, database and business graphics package. The Commodore Plus/4 provides a similar range, although the programs are less sophisticated and really require a disk drive before they can be used effectively. Other micros concentrate on games. Four games are supplied with the Commodore 16, and even Sinclair has started supplying a six-pack of games with its ageing Spectrum.

Anyone buying a new micro should consider several other points. Some machines are more expandable than others, allowing disk drives, printers, modems and other peripherals to be used. Some computers will accept standard add-ons, while others will require 'own-brand' peripherals, which limits the user's choice. A good manual is essential — some machines are supplied with dreadful ones, which confuse more than they enlighten. The prospective buyer should also consider the type of software available for each machine — for example, the BBC Micro has a high proportion of educational software, while the Spectrum is a better bet for games.

Amstrad CPC 464

Amstrad is a company with a wealth of experience in the budget hi-fi market, and this is reflected in the compact design of its first home micro (see page 429). The fact that the machine is supplied with both a monitor and a built-in cassette recorder should make it attractive to the novice user, and the Amstrad features both joystick and Centronics interfaces. There is also a disk drive, which is sold together with the LOGO language and CP/M operating system for £200. The Amstrad is a good, all-round machine, with a maximum graphics resolution of 640 by 200 pixels in two colours, with a 'palette' of 27 shades, and stereo sound facilities. **Price:** £239 (monochrome monitor), £349 (colour monitor)

Sinclair QL

The QL is sold complete with 128 Kbytes of memory, twin built-in Microdrives and four bundled business programs (see page 501). As such, it seems to offer outstanding value — but there are snags. The keyboard is disappointing, being merely a more sophisticated version of the membrane-type keyboard used on the Spectrum; the BASIC is structured but contains a few bugs and is surprisingly slow; long-term reliability of Microdrive storage is also dubious. Editing facilities are also disappointing.

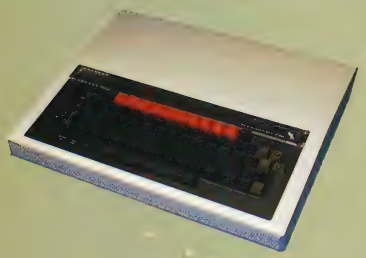
The QL supports graphics resolution of 512 by 256 pixels in four colours, or 256 by 256 in eight. Each pixel may be individually coloured, so 32 Kbytes of memory are used just to handle the screen. No disk or cassette interface is supplied, but the QL does have joystick, monitor, networking and two RS232 interfaces. A major selling point is likely to be the bundled software, written by Psion. These four programs — word processor, database, spreadsheet and business graphics program — are extremely sophisticated when compared with other home machine software. However, they are let down by the QL's hardware limitations, in particular the slow speed of the Microdrives. **Price:** £399

Sinclair Spectrum

The Spectrum has become successful despite its limitations (see page 50). The keyboard is very poor, and the Spectrum uses a 'keyword' system of program entry, which makes programming easier for the beginner but causes problems to the more experienced user. Screen resolution is 256 by 176 pixels with eight colours, two of which may be displayed in any character square. The sound facility is almost non-existent, with a single 'voice' and a volume level that is virtually inaudible. The BASIC is acceptable, although a trifle slow, but the manual serves as a good BASIC tutorial. The Spectrum is entirely lacking in standard interfaces, although many independent firms have produced peripheral equipment that is simply 'hooked on' to the machine's user port. Sinclair recently produced, after a delay of well over a year, its own Interface 1, which supports Microdrives, networking, and an RS232 link. This was quickly followed by the Interface 2, which gave the machine the ability to use cartridge software. Although somewhat outdated, the Spectrum's software base makes it an attractive proposition — especially as six programs, worth over £50, are supplied free with the 48 Kbytes version. **Price:** £100 (16 Kbytes), £130 (48 Kbytes)

BBC Model B

As a result of government recommendation, this machine is widely used in schools and hence should be familiar to younger users; a good range of educational software is available (see page 449). Made by Acorn, the BBC Micro has an excellent specification, with a very fast 'structured' BASIC, excellent graphics resolution, good sound, and an outstanding range of interfaces, including Centronics, RS423, RGB, composite video, four A-to-D channels, a user port, 1 MHz bus for add-ons, and the 'tube', which enables a second processor to be linked to the machine. Disk drives and networking are also supported. However, the relatively small user RAM is rapidly eaten up by the graphics, with between nine and 28 Kbytes left for the user, depending on the mode selected. Use of a second processor alleviates this problem, and selection of the Z80 second processor option enables CP/M software to be run, making the BBC suitable for business use. **Price:** £399

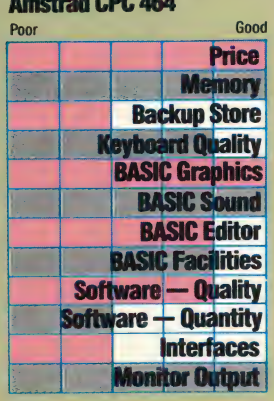


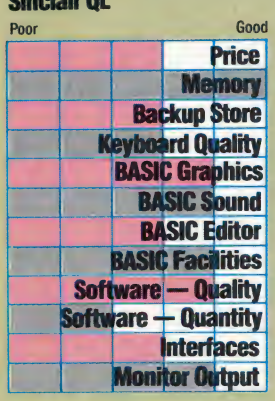
Acorn Electron

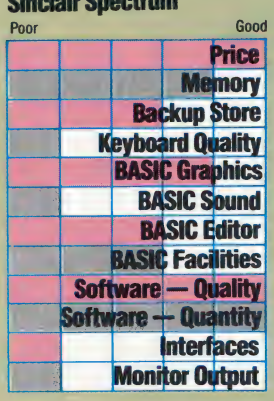
The Electron is a scaled-down version of the BBC Micro, possessing the same excellent structured BASIC, but lacking the wide range of interfaces supplied with the BBC (see page 449). Electron BASIC runs at a slower speed than the BBC version, and the Electron does not support the BBC's Mode 7 teletext graphics. Some BBC software is Electron-compatible, while other programs have been written especially for it. The Electron can produce impressive displays, with a maximum graphics resolution of 640 by 256 pixels, and a maximum text resolution of 32 lines of 80 characters. Unfortunately, this reduces the amount of memory available to the programmer — of the maximum 32 Kbytes, as little as nine Kbytes remains to the user if the highest resolution is selected. The Plus 1 peripheral adds printer, joystick and cartridge interfaces that are not supplied with the basic machine, but as yet no disk drive is available. **Price:** £200

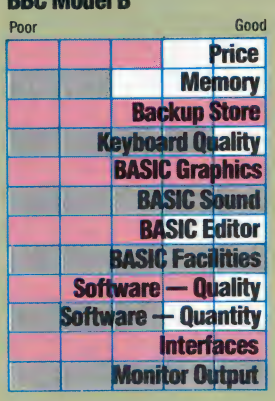
How To Buy A Micro

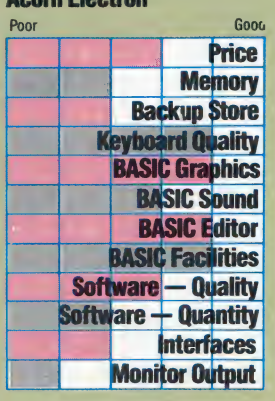
Buying a new micro can be dreadfully confusing. Our chart isolates the most important features of a micro, and produces comparable profiles for the machines illustrated in this article. All you have to do is match your needs to these profiles











Commodore Plus/4

This is the machine that may well eventually replace the Commodore 64. Both machines have a similar 320 by 200 pixel graphics resolution and 64 Kbytes of RAM, but the Plus/4 can display 121 colours and has a much improved BASIC, giving the user more control of the screen display. In the highest resolution mode, two colours only may be displayed in a single character square, but selection of the 160 by 200 pixel mode allows four per square. The sound does not quite match up to the high standards of the Commodore 64, with a maximum of two 'voices'; however, the improved BASIC makes sound-handling considerably easier. The Commodore 64's sprite graphics facility has been omitted in the new model. A machine code monitor is built into the micro, as are four small 'serious' programs — word processor, spreadsheet, database and graphics program. However, a disk drive is needed to make the best use of these. The Plus/4, in line with Commodore policy, requires its own cassette recorder, but this is not the same model as is used on the Vic and Commodore 64. Special joysticks are also required. The Plus/4 uses the same printers and slow disk drive as the Commodore 64, although a faster drive is under development. **Price:** £300

MSX Standard Computers

MSX is a 'minimum standard', and many manufacturers will offer more than the basic specification, although any improvements will not affect compatibility. The machine in our illustration is the Toshiba HX-10 (see page 669). The MSX standard specifies a particularly good version of BASIC, which includes easy-to-use graphics, sound and event-handling commands, as well as a good editor. MSX machines feature function keys, which can be programmed for 10 different functions or commands. The display gives a graphics resolution of 256 by 192 pixels in 16 colours; 32 sprites are also available. To cope with the display, 16 Kbytes of the 80 Kbytes of RAM are reserved for screen-handling. Of the remaining 64 Kbytes, 28 Kbytes may be used by the BASIC programmer; machine code or the use of a disk drive is required to access the remainder. **Price:** around £280

Commodore 64

A well-established micro, with a wealth of available software, the 64 suffers from its poor BASIC, which lacks built-in commands to take advantage of the excellent sound and graphics (see page 10). Maximum resolution is 320 by 200 pixels with 16 colours onscreen, although only two colours may be displayed in each character square. Sprites are also supported. Despite the 64 Kbytes of RAM, no more than 39 Kbytes are available for use. A special cassette recorder is required for use with the C64, and Commodore 'own brand' printers and disk drives are needed if expansion is desired. These are low in cost, but the disk in particular is extremely slow in operation and is prone to errors. **Price:** £200

Atari 600/800 XL

These machines are upgraded versions of the old Atari 400/800 series (see page 189). This means that a wide range of software is available, although the computers themselves now look a little old-fashioned. The Atari 600XL is being phased out, but remains a good buy while stocks last as its 16 Kbyte RAM may be expanded to 64 Kbytes at a cost of £90, effectively turning it into an 800XL. The maximum graphics resolution is 320 by 192 pixels, although two colours only may be displayed in this mode. Selection of a lower resolution permits 16 colours in 16 different shades. Outstanding sound and sprite graphics are other notable features, although the BASIC used by Atari is now somewhat out of date. Atari machines require a special 'dedicated' cassette player, which adds £35 to the price. An annoying feature of Atari micros is the fact that a full manual is not supplied; this must be bought separately. Atari peripherals are non-standard, but are generally widely available and are reasonably priced. **Price:** £100 (16 Kbytes), £170 (64 Kbytes)

Commodore 16

Designed to replace the ageing Vic-20, the Commodore 16 is supplied in a 'starter pack' containing cassette recorder, BASIC tutorial tape and book, and four 'recreational' programs. Although the casing gives the 16 a similar appearance to the Commodore 64 and Vic-20, inside the machine is closer to the £300 Plus/4, using as it does the same BASIC and machine code monitor. In normal use, 12 Kbytes are left for BASIC use, although high resolution graphics will reduce this to a mere two Kbytes. This meagre memory allocation means that most 16 software will be produced on cartridge. Little software has yet appeared for the new machine, but the supplied programs include two arcade games, a chess program-cum-tutor, and a graphics design program. This machine is certainly good value, especially for the novice, but the small amount of free memory could cause problems to software writers. **Price:** £140

Falling Prices

Readers will notice that prices given in this article differ from those quoted in the original reviews: the prices of most machines have dropped significantly in recent months, sometimes by as much as £100

Commodore Plus/4

Poor			Good
			Price
			Memory
			Backup Store
			Keyboard Quality
			BASIC Graphics
			BASIC Sound
			BASIC Editor
			BASIC Facilities
			Software — Quality
			Software — Quantity
			Interfaces
			Monitor Output

Commodore 64

Poor			Good
			Price
			Memory
			Backup Store
			Keyboard Quality
			BASIC Graphics
			BASIC Sound
			BASIC Editor
			BASIC Facilities
			Software — Quality
			Software — Quantity
			Interfaces
			Monitor Output

Commodore 16

Poor			Good
			Price
			Memory
			Backup Store
			Keyboard Quality
			BASIC Graphics
			BASIC Sound
			BASIC Editor
			BASIC Facilities
			Software — Quality
			Software — Quantity
			Interfaces
			Monitor Output

Atari 600/800 XL

Poor			Good
			Price
			Memory
			Backup Store
			Keyboard Quality
			BASIC Graphics
			BASIC Sound
			BASIC Editor
			BASIC Facilities
			Software — Quality
			Software — Quantity
			Interfaces
			Monitor Output

MSX Standard Computers

Poor			Good
			Price
			Memory
			Backup Store
			Keyboard Quality
			BASIC Graphics
			BASIC Sound
			BASIC Editor
			BASIC Facilities
			Software — Quality
			Software — Quantity
			Interfaces
			Monitor Output

CALCULATING MACHINE

Executives and accountants, using expensive 'business' machines, are not the only people who can benefit from a spreadsheet modelling package. This series is designed to provide a practical guide to using cassette-based spreadsheets on home computers.

There are several cheap cassette-based financial modelling packages on the market for popular home computers like the Sinclair Spectrum, Commodore 64 and BBC Model B. This series of articles aims to provide a practical, step-by-step guide to using such spreadsheet packages for a variety of everyday applications – including home budgeting, working out the impact any changes in interest rates might have on mortgage repayments, and comparing the relative merits of leasing and buying major household items.

The heart of any spreadsheet package is an electronic 'worksheet', which is divided into columns and rows (similar to a large ruled sheet of modelling or graph paper). The television screen (or monitor) acts as a movable window that can display any part of this sheet (which is, of course, bigger than the four or five columns and 10 to 15 rows that are shown on the screen at any one time).

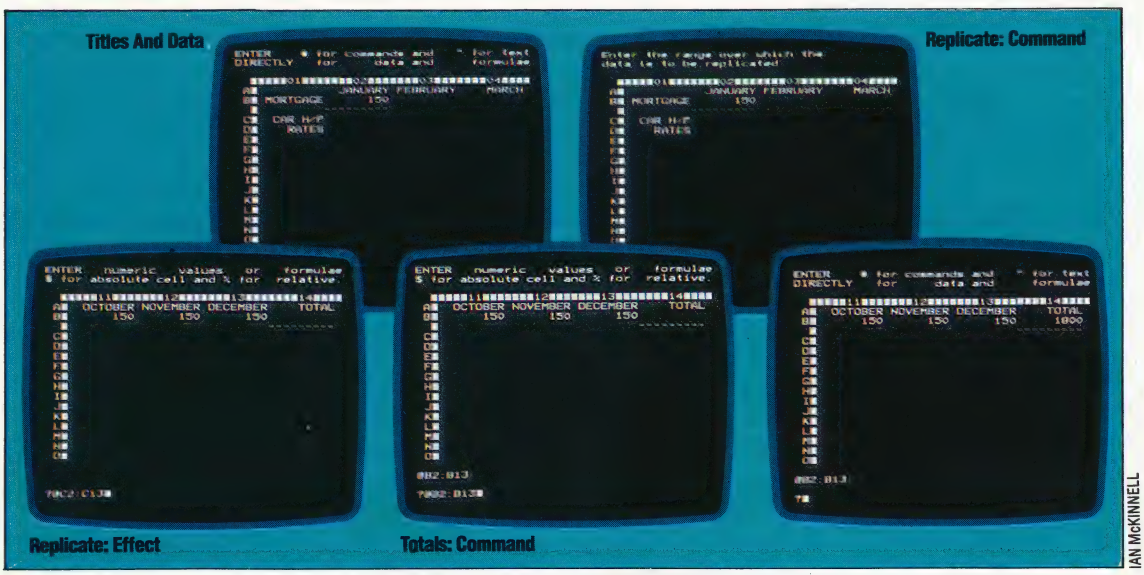
An intersection of a column and row is called a *cell*: each cell can contain numbers, text or formulae. The spreadsheet uses a cursor – normally a highlighted block – that may be moved around it by using the cursor control keys. Any entry of data from the keyboard is assumed by the program to be intended for the cell that is currently occupied by the cursor.

This, then, is a rough outline of the basic design of a spreadsheet. For this first article, we will concentrate on a package called Vu-Calc; this is marketed by Psion and is available for both the Spectrum and the BBC Model B. Here we consider the BBC version, which is almost identical (except for certain minor, but irritating, differences in the names of some commands) to the version for the Spectrum.

Vu-Calc demonstrates just how flexible and useful a very simple spreadsheet can be. It doesn't have any of the more sophisticated features of modelling packages like Lotus 1-2-3 (see page 644). You can't split the screen vertically or horizontally to show different parts of the worksheet at the same time (a facility supplied on the more 'serious' modelling packages), and you don't have anything like the number of features offered by spreadsheets designed for business use. But you can use Vu-Calc to construct some very useful models, which may be saved on cassette, together with your data, for later reference.

The version of Vu-Calc for the BBC Micro has a maximum of 28 columns (numbered from 1 upwards) and 52 rows (labelled alphabetically, with rows after 'Z' allocated double letters – 'AA', 'BB', etc.). This is not very large by spreadsheet standards, but the limit is set by the BBC Micro's rather meagre 32 Kbyte memory.

One of the most useful applications for a home computer spreadsheet is the annual household budget, which also has the merit of being a relatively simple model to build. Once constructed, such a model enables you to see at a glance what the impact of any increased expenditure — an unduly heavy telephone bill,



IAN McKINNELL

say, or an unscheduled trip to France — will be on your expected cash surplus (always assuming that you have one). In other words, once the model is built, you can play around with the data and see how the effect of one alteration can be reflected across the spreadsheet as a whole.

BUILDING A MODEL

The first step in building such a model is to write down on a piece of paper a list of all the household expenses that you can think of. The next task is to note the expected monthly figure under each heading. It is here, at the point of calculating or estimating the monthly amount and keying it into the model, that Vu-Calc's strengths become apparent. The expense categories are keyed in one after the other down the first column, and each subsequent column is labelled 'Jan', 'Feb', 'Mar', etc. Vu-Calc requires that all text entries be preceded by a double quotation mark, but this punctuation does not appear on the displayed spreadsheet. The first few rows and columns of our model might therefore look like this:

	1	2	3	4
A		JAN	FEB	MAR
B	Car H/P			
C	Mortgage	150		
D	Rates			

This gives us the basic 'shape' of the model. We now need to input the relevant values, and to do this we use Vu-Calc's REPLICATE command.

Vu-Calc has a limited number of commands, all of which are prefixed with a # sign. (When you type # at a blank cell, the program goes into 'command mode' and waits to be told the command you want to use.) The most useful of these commands is REPLICATE – this is because the most boring part of building a model is the need to key in all the values the model will use. REPLICATE is basically a labour-saving device that allows the same piece of data to be entered into many different cells simultaneously.

A common part of a household expenses/budget model is the fixed monthly expenses, such as the rates, the mortgage or the rent. If the mortgage is, say, £150 a month, the REPLICATE command would be used to insert £150 in the appropriate 12 cells.

When you invoke the REPLICATE command by typing #R, a prompt line appears at the top of the screen, above the model itself, which reads:

> Replicate – Enter the cell to replicate, RETURN for the current cell.

This asks you to specify the cells that are to be copied (notice that you can copy only one cell, not a whole block of cells – this is one of Vu-Calc's more obvious limitations). The cell is specified by its co-ordinates, with the row letter given first, followed by the column number – i.e. C2. Once this has been entered, the prompt line asks you to:

> Enter the range over which the data is to be replicated.

A range of cells in Vu-Calc is indicated by specifying the first (or leftmost) cell and the bottom rightmost cell of the range. (Think of the block of cells as a box; you have to tell the program the co-ordinates of the box's top left and bottom right corners.)

In our example, we want to tell the program to put £150 into the range of cells in Row C, from C3 to C13 (the columns labelled 'Feb' to 'Dec'). The format for this is #R,C2,C3:C13. This automatically fills each cell, almost instantly, with the value 150. (The real power of the REPLICATE command comes in copying formulae from cell to cell, but this is a fairly specialised area, since such formulae can either be 'relative' to a particular cell or 'absolute' – a distinction and a topic that we will cover in the next instalment.)

We now work through all our expense headings in the same way. Note that if you decide that in specified months the amount for a particular expense should be greater or less than the standard value, you can simply move the cursor to that cell and enter a new value. This will immediately overwrite the old figure.

In our model, column 14 will be the monthly totals column. There would be little point in using a spreadsheet if you needed to use a calculator to add up each month's expenses, so Vu-Calc can be used to add all values in any row or column. The @ command indicates that you want to add the values contained in a range of cells; this range is specified in the same way as before. So, to total the year's mortgage payments, and to put the result in cell C14, you simply place the cursor at C14 and type @C2:C13. The result – 1800 in this case – is immediately displayed.

We could just as easily have put a formula into cell C14; instead of adding all the values we could have entered C2*12. Vu-Calc would have taken this to be a formula, as the C is not preceded by a quote mark, and would have executed it immediately, giving the result 1800. This illustrates that there is often more than just one way to achieve a particular result.

In the next instalment of the course, we will look at how expenses can 'grow' by fixed percentages, and how relative and absolute cell reference formulae may be replicated.

Spreading It Around

Commodore 64
Busicalc: £45. Cassette/disk by Supersoft, Winchester House, Canning Road, Harrow HA3 7SJ.
Insta-Calc Graphic: £87.50. Cartridge/disk by Dataview Wordcraft Ltd., Radix House, East Street, Colchester CO1 2XB.

BBC Micro
VuCalc: £10.Cassette by Psion Ltd., 2 Huntsworth Mews, Gloucester Place, London NW1 6DD.

Spectrum
VuCalc: £10.Cassette by Psion Ltd.

DRAGON SLAYER

In the last instalment of the course we published four sprite bit maps suitable for use with Commodore 64 LOGO. Continuing our investigation of its sprite facilities, we develop the 'curve of pursuit' algorithm through the three bugs problem and a pursuit game using the sprites, and a sophisticated interception strategy.

We give here the procedures for a game that uses LOGO sprites. You control a dragon that attempts to reach and destroy a city. The defence of the city is in the hands of a flying knight (under the control of the computer), who will try to kill the dragon. You control the dragon's direction of movement with the joystick. If you do evade the knight and get close enough, the city will burst into flames from the dragon's breath.

To run the game you will need to read in the SPRITES file, define your shapes, type in the procedures and then type GAME. After performing various set up tasks, the GAME procedure then calls PLAY, which is the central procedure. PLAY moves the dragon and the knight in turn, and checks to see if the dragon has reached the city, or if the knight has hit the dragon. The remaining procedures carry out other parts of PLAY's actions.

The available colour commands are very straightforward. To set the background colour use BACKGROUND followed by a colour number, and to set a sprite colour (and the colour of the line it draws if the pen is down) use PENCOLOR. The colour numbers are given names in INIT. VARIABLES, so that we can then specify colours by name, using commands such as PENCOLOR :RED.

In the procedure PLAY, the line:

```
IF HIT? THEN DRAGON. DESTROYED
```

is used to test if the knight has hit the dragon. The procedure HIT? illustrates the way in which we can write our own test conditions in LOGO. It returns a value of "TRUE or "FALSE, and this is used as an input to the IF statements. The result "TRUE would then cause the conditional action to be carried out.

HIT? uses a procedure from the SPRITES file, TS?, which returns "TRUE if a sprite is touching the

Knights And Dragons

The Game In Progress

City In Flames

Dragon Defeated

IAN McKINNELL

Knight Vs Dragon

```
TO GAME
   INIT. VARIABLES
   SET.SCREEN
   PLAY
END

TO INIT.VARIABLES
   MAKE "FLAME 1
   MAKE "FLAME1 2
   MAKE "DRAGON 3
   MAKE "KNIGHT 4
   MAKE "CITY 5
   MAKE "CITY1 6
   MAKE "RED 2
   MAKE "BLACK 0
   MAKE "BLUE 6
   MAKE "ORANGE 8
   MAKE "YELLOW 7
END

TO SET.SCREEN
   DRAW
   FULLSCREEN
   BACKGROUND :BLUE
   TELL 0
   PU
   HT
   TELL :FLAME1
   HT
   HT
   TELL :FLAME
   HT
   POSITION :DRAGON 100 100 :RED
   BIGX BIGY
   POSITION :CITY ( - 52) ( - 80) :BLACK
   BIGX BIGY
   POSITION :CITY1 ( - 100 ) ( - 80 ) :BLACK
   BIGX BIGY
   TELL 4
   PU
   POSITION :KNIGHT ( 100 ) ( - 100 ) :YELLOW
   SMALLX SMALLY
END

TO PLAY
   DRAGON.MOVE
   IF DISTANCE :DRAGON :CITY < 50 THEN CITY.
      DESTROYED STOP
   IF DISTANCE :DRAGON :CITY1 < 50 THEN CITY.
     DESTROYED STOP
   KNIGHT.MOVE
   IF HIT? THEN DRAGON.DESTROYED STOP
   PLAY
END

TO DRAGON.MOVE
   TELL :DRAGON
   MOVEJOY JOYSTICK 1
   FD 10
END

TO MOVEJOY :DIR
   IF :DIR < 0 STOP
   SETH :DIR * 45
END

TO DISTANCE :A :B
   TELL :A
   MAKE "X1 XCOR
   MAKE "Y1 YCOR
   TELL :B
```

current sprite. HIT? first sets the current sprite to the dragon and then asks if anything is touching it.

The command JOYSTICK takes 0 or 1 as input (corresponding to ports 1 and 2). The output is –1 if the joystick is central, 0 if up, 1 if at 45°, 2 if at 90°, and so on up to 7. Here, we simply set the dragon's heading to 45° multiplied by the output number.

Explosions and similar effects are easy to achieve using sprites. A shape representing the explosion is simply flashed on top of the object to be destroyed. We give the FLAME sprite a low number so that it will have high priority and thus appear on top of the other sprites.

The computer controls the knight, but it uses a very simple defensive strategy: the knight heads straight towards the dragon. As the game stands the dragon can slip by the knight and destroy the town fairly easily.

How can we improve the knight's defensive strategy? One simple way is to increase his speed, and simply increasing this from 10 to 11 makes it very difficult for the dragon to get by. (Wrapping round the screen is cheating!) A slightly better strategy is for the knight to aim to cut the dragon off by heading for the line between the dragon and the city and staying there.

In the next instalment, we will look at some of the sprite features found in Atari LOGO.

Three Bugs Project

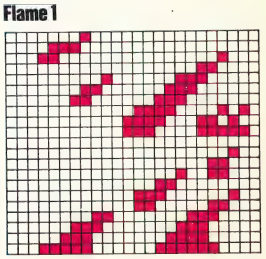
The bugs start out from the corners of a triangle:

```
TO BUGS
   SETUP MOVE.BUGS
END

TO SETUP
   DRAW FULLSCREEN TELL 0 HT PU SETXY
   ( – 100 ) ( – 100 ) TRI 200 POSITION 1 ( – 100 )
   ( – 100 ) POSITION 2 0 73 POSITION 3 100
   (–100)
END

TO TRI :SIDE
   PD REPEAT 3 [FD :SIDE RT 120] PU
END

TO POSITION :NO :X :Y
   TELL :NO SETSHAPE 3 PU SETXY :X :Y PD ST
END

TO MOVE.BUGS
   FOLLOW 1 2 FOLLOW 2 3 FOLLOW 3 1
   MOVE.BUGS
END

TO FOLLOW :A :B
   TELL :B MAKE "X XCOR MAKE "Y YCOR TELL :A
   SETH TOWARDS :X :Y FD 10
END
```

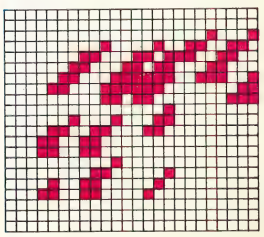
Sprite Editor

```
   MAKE "X2 XCOR
   MAKE "Y2 YCOR
   OUTPUT SQRT ( ( :X1 – :X2 ) * ( :X1 – :X2 ) +
      ( :Y1 – :Y2 ) * ( :Y1 – Y2 ) )
END

TO CITY.DESTROYED
   TELL :CITY
MAKE "X XCOR
   MAKE "Y YCOR
   FLASH :X :Y :ORANGE
   TELL :CITY1
   MAKE "X2 XCOR
   MAKE "Y2 YCOR
   FLASH1 :X2 :Y2 :ORANGE
   HIDE.SPRITE
   SPLITSCREEN
   REPEAT 3 [PRINT "]
   PRINT [CITY DESTROYED!]
END

TO FLASH :X :Y :COLOR
   TELL :FLAME
   PENCOLOR :COLOR
   SETXY :X :Y
   ST
   REPEAT 6 [SMALLX SMALLY WAIT BIGX BIGY
      WAIT]
END

TO FLASH1 :X2 :Y2 :COLOR
   TELL :FLAME1 PENCOLOR :COLOR
SETXY :X2 :Y2
   ST REPEAT 6 [SMALLX SMALLY WAIT BIGX BIGY
      WAIT]
END

TO KNIGHT.MOVE
   TELL :DRAGON
   MAKE "X XCOR
   MAKE "Y YCOR
   TELL :KNIGHT
   SETH TOWARDS :X :Y
   FD 10
END

TO HIT?
   TELL :DRAGON
   IF TS? THEN OUTPUT "TRUE
   OUTPUT "FALSE
END

TO DRAGON.DESTROYED
   TELL :DRAGON
   MAKE "X XCOR
   MAKE "Y YCOR
   FLASH :X :Y :BLACK
   HIDE.SPRITE
   SPLITSCREEN
   REPEAT 3 [PRINT "]
   (PRINT [DRAGON KILLED AT A DISTANCE OF]
      DISTANCE :DRAGON :CITY )
END

TO HIDE.SPRITE
   TELL :FLAME HT
   TELL :FLAME1 HT
   TELL :CITY HT
   TELL :CITY1 HT
END

TO WAIT
   REPEAT 100 [ ]
END
```

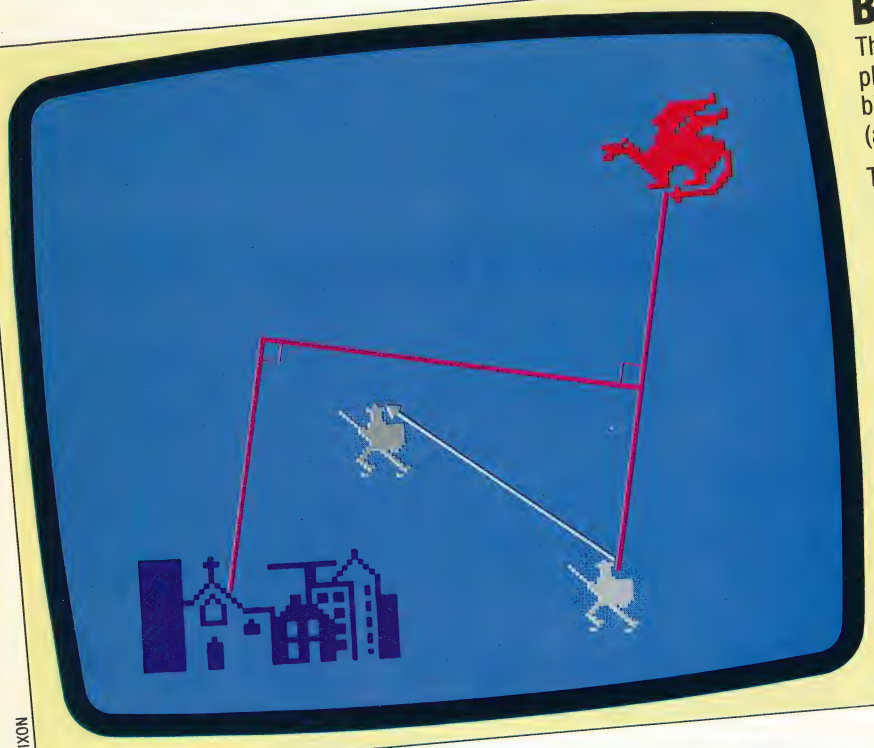
Flame 1



Flame 2



LIZ DIXON



LIZ DIXON

Best Strategy

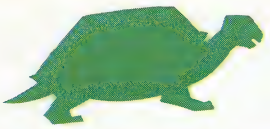
The best strategy (for both parties) is to head for the place closest to the city on the perpendicular bisector of the line joining the knight and the dragon (as shown in the diagram):

```
TO KNIGHT.MOVE
   TELL :DRAGON MAKE "DX XCOR MAKE "DY
   YCOR TELL :KNIGHT MAKE "KX XCOR MAKE
   "KY YCOR TELL :CITY MAKE "CX XCOR MAKE
   "CY YCOR
   MAKE "SX ( :DX + :KX ) / 2
   MAKE "SY ( :DY + :KY ) / 2
   MAKE "VX ( :DY – :KY )
   MAKE "VY ( :KX – :DX )
   MAKE "FACT ( :VX * (:CD – :SX ) + :VY *
   ( :CY – :SY ) ) / ( ( :VX * :VX) * ( :VX * :VX ) +
   ( :VY * :VY ) * ( :VY * :VY ) )
   MAKE "X :SX + :FACT * :VX
   MAKE "Y :SY + :FACT * :VY
   TELL :KNIGHT SETH TOWARDS :X :Y
   FD 10
END
```

Improved Move

An improved knight move, in which the knight heads to cut off the dragon:

```
TO KNIGHT.MOVE
   TELL :DRAGON MAKE "X XCOR MAKE "Y YCOR
   TELL :CITY
   MAKE "BEARING TOWARDS :X :Y
   TELL :KNIGHT SETH 270 + :BEARING
   IF XCOR < :X THEN LEFT 180
   FD 10
END
```



Abbreviations

BACKGROUND	BG
PENCOLOR	PC

Keyboard Control

Keyboard control of the dragon:

```
TO DRAGON.MOVE
   TELL :DRAGON MOVE READKEY FD 10
END

TO MOVE :DIR
   IF :DIR = "W THEN SETH 0
   IF :DIR = "S THEN SETH 90
   IF :DIR = "Z THEN SETH 180
   IF :DIR = "A THEN SETH 270
END

TO READKEY
   IF RC? THEN OUTPUT READCHARACTER
   OUTPUT "
END
```

Logo Flavours

Spectrum LOGO and the Apple LOGOs don't feature sprite graphics.

Atari users should note the following differences:
1) TS? does not exist. Omit the line IF HIT? etc. in PLAY and insert this line as the last line of SET.SCREEN:

```
   WHEN TOUCHING :DRAGON :KNIGHT
   [DRAGON.DESTROYED STOP]
```

2) There are no equivalents for BIGX, BIGY, SMALLX and SMALLY. Simply omit them.
3) For BACKGROUND use SETBG, and for PENCOLOR use SETPC. The colour codes are different, of course.
4) Most surprisingly TOWARDS does not exist in Atari LOGO (It is included in the LCSI versions on the Apple and the Spectrum). Therefore, replace the lines:

```
   SETH TOWARDS :X :Y
   FD 10
```

in FOLLOW and KNIGHT.MOVE with

```
   MAKE "FRAC 10 / (SQRT ( ( XCOR – :X ) * ( XCOR
   – :X ) + ( YCOR – :Y ) * ( YCOR – :Y ) ) )
   SETPOS LIST ( XCOR + ( :X – XCOR ) * :FRAC )
   ( YCOR + ( :Y – YCOR ) * :FRAC )
```

SUSPENDING OPERATIONS

We briefly introduced the concept of 'interrupt handling' when we reviewed the Toshiba HX-10 (see page 669). These are messages that interrupt the task that a processor is currently performing, in order to convey important information to it. Here, we explore the interrupt mechanism in detail.

One common application of interrupts is when we are dealing with input from the keyboard. If a program directly accesses the keyboard — usually via the operating system — to obtain the next input character, then any key that is pressed while the program is doing something else will be lost. Even when the processor is fully engaged in processing keyboard input, it is still possible for it to lose a character, especially one that follows a character that needs extra processing, such as a carriage return.

The solution is for the keyboard to interrupt the processor whenever a key is pressed, so that the processor stops what it is doing and performs an 'interrupt service routine'. This takes the character that has just been input and places it in a section of memory reserved as a *keyboard buffer*. The processor can then return to whatever it was doing and carry on as though nothing has happened.

Whenever the operating system keyboard input routine is called, it does not look at the keyboard directly but takes the next character out of the buffer instead (waiting for one to appear if the buffer is empty). This mechanism enables the user to 'type ahead' of what actually appears on the screen, and should ensure that no characters are lost.

There are, however, two possible problems. The user may type so quickly that the buffer fills up faster than the program can deal with the input, thus causing the buffer to overflow. The solution to this requires a compromise between allowing sufficient memory for an adequate-sized buffer and not wasting too much valuable memory space. The second problem arises with those users who feel uncomfortable when a character does not appear on the screen immediately a key is pressed. They may keep pressing the key, and thereby generate dozens of characters that then go into (and again may overflow) the buffer. This problem is usually solved by familiarity with the computer.

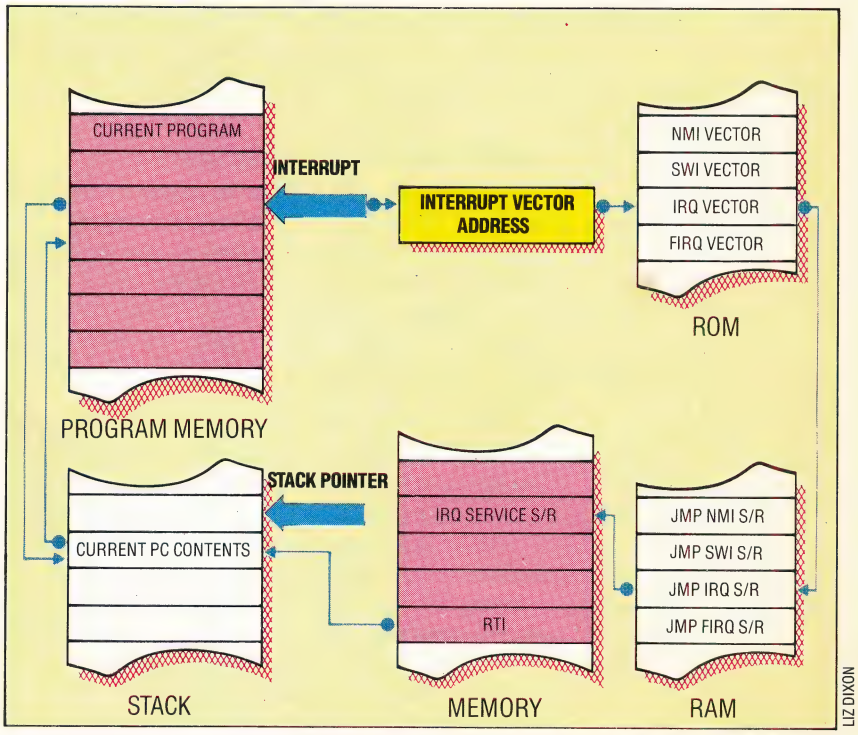
Another useful application for interrupts occurs when output is sent to a printer — which is often one of the most time-consuming operations a micro needs to perform. During printing, the processor may be required to work for 100 microseconds while it sends a character to the printer, and then wait thousands of microseconds for the printer to process that character. A *spooling* system is one answer to this: it places the files to be printed in a queue, and part of the first file in the queue is loaded into another buffer area of memory. The port that serves the printer will interrupt the processor whenever the printer is ready for another character to be sent. The interrupt service routine will then send the next character from the buffer, or (if the buffer is empty) load the next section from the file at the head of the queue into the buffer. In this way, printing can be going on in the background, while the processor is free to get on with something else.

TYPES OF INTERRUPT

There are some operations that the processor performs — such as accessing disks — where being interrupted can cause data loss or some other catastrophe. There must be, therefore, a mechanism for *masking* interrupts so that the processor can ignore any that occur during a particularly sensitive operation. If this is the case, it is preferable that a note is made to indicate that the interrupt has occurred, so that it can be dealt with later.

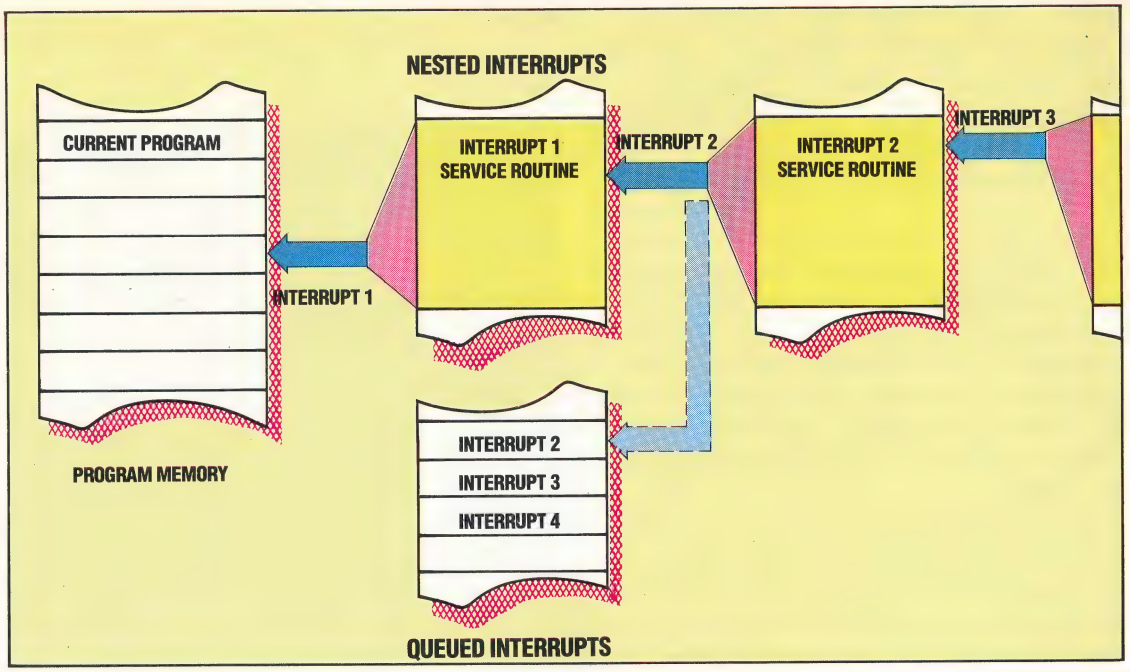
Conversely, if we are dealing with a disk interface that is interrupt-driven, then its

Interrupt One
When an interrupt occurs, the processor completes execution of the current instruction, and stacks the current contents of the program counter. The interrupt vector address appropriate to the interrupt is then loaded into the program counter, and control passes to that address — usually in ROM. This address points to another address — usually in RAM — where a JMP instruction directs control to the actual interrupt service routine. This is terminated by the RTI instruction, which passes control back to the main program via the return address on the stack. Since the JMP instruction is stored in RAM, it can be found and changed by the programmer so that on an interrupt, control passes first to a special-purpose user routine and then to the normal service routine



Interrupted Interrupts
If an interrupt occurs during an interrupt, one possible solution is for the processor to 'nest' the interrupts: whenever an interrupt occurs, PC is stacked and the new interrupt is handled immediately and control returns to the stacked address. The limits on this nesting are the capacity of the stacks, and the ability of the interrupt-generating devices to withstand delays in processing their interrupts.

An alternative is for the processor to stack the details of any interrupt on an interrupt queue. When the first interrupt is complete, the processor inspects the queue, processes any interrupt it finds there until the queue is empty, and eventually passes control back to the program



interrupts would be given priority and under no circumstances masked — thus giving rise to the concept of a *non-maskable* interrupt. Such an interrupt might come from a circuit that detects a drop in the mains supply voltage: its service routine would immediately start saving the current task while power remained.

When interrupts can occur from more than one source, we must consider the possibility of *nested* interrupts. If an interrupt occurs while the processor is in the middle of servicing another interrupt, there are two possible strategies for handling it. First of all, the new interrupt could be ignored until the current one is completed. Secondly, interrupts could be ranked on a scale of urgency, so that a high-priority interrupt could override the handling of one with a lower priority. In this case, the operating system would have to be able to deal with the nesting of interrupt service routines.

SOFTWARE INTERRUPTS

The SWI instruction, which we briefly mentioned on page 577, can be used in a program as a convenient way of returning to the operating system by generating its own interrupt — called the 'software interrupt' (as distinct from the hardware-generated interrupts we have been discussing so far). SWI instructions can also be used to act as breakpoints in a machine code program to aid in debugging; this facility is provided by many ROM-based machine code monitors, as well as debugging packages. The user chooses points in the code where program execution is to pause, and the instructions at these locations are replaced with SWIs. When the program is run, the interrupt service routine then allows the programmer to inspect and possibly alter the contents of registers and memory locations, and see exactly what the program is doing. When execution is resumed, the monitor/debugger replaces the instruction displayed by the breakpoint SWI, and continues with the program from that point.

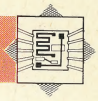
The 6809 has three separate interrupt mechanisms: IRQ (Interrupt ReQuest), FIRQ (First Interrupt ReQuest) and NMI (Non-Maskable Interrupt). These are all activated by the appropriate signal being received on three pins on the processor chip. The bar above the name (in $\overline{\text{IRQ}}$, for example) indicates that they are activated by a zero signal at the processor, rather than a one. These three pins are connected to the main bus so that peripheral chips like the 6820 and 6850 can have their interrupt request output pins connected to the same bus lines. When the chips are programmed, the interrupts can be enabled and then the appropriate signals will automatically be sent.

There are also three software interrupts caused by the SWI, SW12 and SW13 instructions.

When an interrupt occurs, control passes to the *vector* address contained in a specific location at the top of memory. These vector addresses are usually found in ROM, so control will always pass from there to the same fixed address. However, this address will normally be in RAM and will contain a JMP instruction, so that the final destination can be changed to the user's own service routine. The memory locations are:

Interrupt Type	Vector
$\overline{\text{NMI}}$	$FFFC
SWI	$FFFA
$\overline{\text{IRQ}}$	$FFF8
$\overline{\text{FIRQ}}$	$FFF6
SW12	$FFF4
SW13	$FFF2

It is also worth noting that the top two bytes of memory — $FFFE and $FFFF — contain the *reset*



vector, the address to which control is transferred on power-up or a hardware reset; this is usually the start address of the ROM monitor. Furthermore, the two bytes at $FFF0 and $FFF1 are reserved by Motorola for possible future use.

Information about interrupts is contained in three bits of the condition code register (CC): bit 4 (I), bit 6 (F) and bit 7 (E). Setting the I bit to one masks the IRQ interrupt, setting the F bit masks FIRQ. The E bit is used by the processor to differentiate between IRQ or NMI, and FIRQ: if E gets set to one then there has been an IRQ or NMI, if E gets set to zero then an FIRQ has occurred.

When an interrupt is received, it is treated in a similar way to a subroutine call: the contents of some or all of the registers are stacked so that control can be returned to the same point in the program currently being executed. The interrupt service routine ends with an RTI instruction (similar to an RTS), which unstacks the registers and returns control to the original program.

The actual difference between the FIRQ and the other two interrupts is that FIRQ stacks the program counter (PC) and the condition code (CC) registers only, and is therefore much faster in operation than the other two. The interrupt service routine, however, must restore any registers that it uses, so this type of interrupt should not be used for routines that use more than just one or two registers. We can see now where the E bit is used because the same RTI instruction is used to terminate IRQ and FIRQ routines, but the processor must determine which registers need unstacking. The sequence of events when an interrupt occurs is:

1) The current instruction is executed.
2) The I bit is set, disabling other IRQ interrupts. If the interrupt was a FIRQ or a NMI then bit F is also set to disable FIRQ. SW12 and SW13 do not mask other interrupts, but SW1 does.
3) On an FIRQ bit E is cleared to zero, otherwise it is set to one.
4) The vector in the appropriate memory locations is loaded into PC and execution continues from that address.

Our first program looks at another good use for interrupts; namely maintaining a real-time clock. We shall assume that some timing device, which could be a special purpose chip like the 6840 interval timer or a division of the system clock or a modification of the 50Hz mains, is connected to a PIA at $5000. The first subroutine will enable the interrupts and set up a 16-bit counter at $50. The interrupt service routine will simply increment the counter so that at any time inspection of $50 will give the number of timing signals that have been received, from which the time can be calculated if the start-up time and the frequency of the timing signals are known.

The second example program assumes a printer is connected to the same PIA at $E000. We shall employ a buffer, of indeterminate length, at $100 to store one line of output to be printed by the service routine. A flag at $50 is set to zero while the line is being printed, and to one when the line is finished. This will enable some other routine (which we shall not be concerned with here) to refill the buffer. Locations $51 and $52 contain a pointer into the buffer giving the address of the next character to be printed. The first subroutine sets up the PIA, flag and buffer pointer for a new line.

Program One

```
PIACR   EQU   $E001          PIA control register
PIADR   EQU   $E000          PIA data register
INTRP   EQU   $2000
CLK1    EQU   $50
CLK2    EQU   $51
        ORG   $1000          Subroutine to initialise clock
INITCK  CLR   CLK1           Clear clock locations
        CLR   CLK2
        LDA   #%00000101     Enable PIA interrupts
        STA   PIACR
        ANDCC #%11101111     Enable IRQ
WAITCK  TST   CLK2           Wait for first increment
        BEQ   WAITCK
        RTS
        ORG   INTRP          Interrupt service routine
        LDA   PIADA          Clear interrupt
        LDD   CLK1           Get count
        ADDD  #1             Increment count
        STD   CLK1
        RT1
```

Program Two

```
PIACR   EQU   $E001          PIA control register
PIADR   EQU   $E000          PIA data register
INTRP   EQU   $2000
CR      EQU   13             Carriage return
BUFFER  EQU   $100           Address of buffer
BUFPTR  EQU   $51            Buffer pointer
FLAG    EQU   $50            End-of-line flag
        ORG   $1000          Subroutine to set everything up
        CLR   FLAG           Clear end-of-line flag
        LDX   #BUFFER        Initialise buffer pointer
                             to start of buffer
        STX   BUFPTR
        CLR   PIACR          Address data direction register
        LDA   $FF            Set all lines to output
        STA   PIADR
        LDA   #%00000101     Enable PIA interrupts
        STA   PIACR
        ANDCC #%11101111     Enable IRQ
        RTS
        ORG   INTRP          Interrupt service routine
        LDX   BUFPTR         Buffer pointer
        LDA   ,X+            Get next character from buffer
        STA   PIADR          Print it
        LDB   PIADR          Clear interrupt
        STX   BUFPTR         Incremented buffer pointer
        CMPA  #CR            Was it end-of-line?
        BNE   FINISH         Skip if not end-of-line
        INC   FLAG           Else set flag
FINISH  RTI
```

CALLING COMMODORE

The variable search program that we developed for the BBC Micro and the Spectrum on pages 664 to 665 can easily be converted to work on the Commodore 64. The Commodore 64 program is, in fact, a little simpler because there are not so many special cases to allow for.

Many of the variable names in the Commodore version of the program have to be abbreviated to avoid including a BASIC keyword. For example, NEWLINE cannot be used as a variable name because it starts with NEW, and TEXTPOINTER cannot be used either because it includes INT.

The changes near the beginning of the program are necessary because of the differences in the way a line of BASIC is stored in the computer's memory. In the BBC Micro and the Spectrum, a line of BASIC in the internal format begins with a two-byte line number, with the high order byte coming first, and one or two bytes for the length of the line. In the Commodore 64 a line of BASIC begins with a two-byte pointer to the start of the next line and a two-byte line number, with the low order byte coming first in both.

We still have to skip REM lines and strings inside quotes, but we do not have to look for any other special cases that might cause confusion, like the hexadecimal numbers on the BBC Micro or the hidden binary form of numbers on the Spectrum.

The section of the program that actually picks out the variable names looks for a letter of the alphabet first, then letters or digits, and at the end it looks for a $ or % sign indicating a string or integer variable and a (, indicating a function or array. The Commodore 64 does not allow the underscore character that can be included in variable names on the BBC Micro and the Spectrum.

Although the Commodore 64 can display both upper and lower case letters, the difference is only in the form of the character as it appears on the screen, and not in the internal code for the character. Thus, the program only needs to look for upper case letters in a variable name.

The Commodore 64 version of the program is used in the same way as the BBC Micro and Spectrum versions. Type in the search program and SAVE it, then LOAD the program to be searched and append the search program to it. You can then search the program by "RUN 30000", and typing in the variable name when the program asks for it, ending the name with "(" if you want to find an array name.

There is a simple method of joining two SAVEd programs on the Commodore 64, provided the line numbers in the first program are all less than the line numbers in the second program. The method uses two of the pointers in page zero: TXTTAB, at addresses 43 and 44, which hold the address where the BASIC program starts, and VARTAB, at addresses 45 and 46. A BASIC program ends with a byte containing zero that marks the end of the last line of the program, then two more zero bytes that mark the end of the program. The address in VARTAB is normally the byte following the last of these zeros. To join two programs together, first LOAD the program with the lowest line numbers, then type:

```
PRINT PEEK(45), PEEK(46)
```

If the first number is between two and 255 subtract two from it and POKE the result into address 43. If it is zero or one, POKE 254 or 255 into address 43 and POKE one less than the result of PEEK(46) into address 44. You can then LOAD the second program, and finally type:

```
POKE 43,1: POKE 44,8
```

This will put the normal value back into the 'start of BASIC' pointer and the programs will now be joined together.

Commodore 64 Search Utility

```
30000 INPUT "NAME TO SEARCH FOR"; T$
30010 RE=143
30020 QUOTE=34
30030 NL=0
30040 TEXTPTR=2049
30050 NXTLINE=PEEK(TEXTPTR)+256*PEEK(TEXTPTR+1)
30070 TEXTPTR=TEXTPTR+2
30080 LINENO=PEEK(TEXTPTR)+256*PEEK(TEXTPTR+1)
30085 IF LINENO>=30000 THEN END
30090 TEXTPTR=TEXTPTR+2
30140 IF PEEK(TEXTPTR)=NL THEN TEXTPTR=TEXTPTR+1:GOTO 30050
30150 IF PEEK(TEXTPTR)<>RE THEN GOTO 30180
30160 REM SKIP OVER REM LINE
30170 TEXTPTR=NXTLINE:GOTO 30050
30180 IF PEEK(TEXTPTR)<>QUOTE THEN GOTO 30300
30190 REM SKIP ANYTHING IN QUOTES,STOP AT END OF LINE IN CASE OF UNMATCHED
      QUOTE
30200 TEXTPTR=TEXTPTR+1
30210 IF PEEK(TEXTPTR)=NL THEN TEXTPTR=TEXTPTR+1:GOTO 30500
30220 IF PEEK(TEXTPTR)<>QUOTE THEN GOTO 30200
30230 TEXTPTR=TEXTPTR+1
30235 GOTO30140
30290 REM FIRST CHARACTER OF NAME MUST BE LETTER
30300 IF PEEK(TEXTPTR)>=ASC("A") AND PEEK(TEXTPTR)<=ASC("Z") THEN GOTO 30330
30310 TEXTPTR=TEXTPTR+1
30320 GOTO30140
30330 NAME$=""
30340 NAME$=NAME$+CHR$(PEEK(TEXTPTR))
30350 TEXTPTR=TEXTPTR+1
30360 REM LETTER OR DIGIT AFTER FIRST CHARACTER
30370 IF PEEK(TEXTPTR)>=ASC("A") AND PEEK(TEXTPTR)<=ASC("Z") THEN GOTO 30340
30390 IF PEEK(TEXTPTR)>=ASC("0") AND PEEK(TEXTPTR)<=ASC("9") THEN GOTO 30340
30410 REM END WITH $ FOR STRING VARIABLE, % FOR INTEGER VARIABLE
30420 IF PEEK(TEXTPTR)=ASC("$") THEN NAME$=NAME$+"$":TEXTPTR=TEXTPTR+1
      :GOTO30450
30430 IF PEEK(TEXTPTR)=ASC("%") THEN NAME$=NAME$+"%":TEXTPTR=TEXTPTR+1
30440 REM ( IF ARRAY OR FUNCTION
30450 IF PEEK(TEXTPTR)=ASC("(") THEN NAME$=NAME$+"(":TEXTPTR=TEXTPTR+1
30460 IF NAME$=T$ THEN PRINT NAME$;" IN LINE";LINENO
30470 GOTO 30140
30480 END
```

DATABASE

Here, courtesy of Zilog Inc., we produce another part of the Z80 programmers' reference card.

16-Bit Arithmetic Group

DESTINATION		SOURCE	BC	DE	HL	SP	IX	IY
	'ADD'	HL	09	19	29	39		
		IX	DD 09	DD 19		DD 39	DD 29	
		IY	FD 09	FD 19		FD 39		FD 29
	ADD WITH CARRY AND SET FLAGS 'ADC'	HL	ED 4A	ED 5A	ED 6A	ED 7A		
	SUB WITH CARRY AND SET FLAGS 'SBC'	HL	ED 42	ED 52	ED 62	ED 72		
	INCREMENT 'INC'		03	13	23	33	DD 23	FD 23
	DECREMENT 'DEC'		08	1B	2B	3B	DD 2B	FD 2B

Mnemonic	Symbolic Operation	S	Z		H		P/V	N	C	Opcode 76 543 210	Hex	No.of Bytes	No.of M Cycles	No.of T States	Comments
ADD HL, ss	HL ← HL + ss	•	•	X	X	X	•	0	↕	00 ss1 001		1	3	11	ss Reg 00 BC
ADC HL, ss	HL ← HL + ss + CY	↕	↕	X	X	X	V	0	↕	11 101 101 01 ss1 010	ED	2	4	15	01 DE 10 HL 11 SP
SBC HL, ss	HL ← HL − ss − CY	↕	↕	X	X	X	V	1	↕	11 101 101 01 ss0 010	ED	2	4	15	
ADD IX, pp	IX ← IX + pp	•	•	X	X	X	•	0	↕	11 011 101 01 pp1 001	DD	2	4	15	pp Reg 00 BC 01 DE 10 IX 11 SP
ADD IY, rr	IY ← IY + rr	•	•	X	X	X	•	0	↕	11 111 101 00 rr1 001	FD	2	4	15	rr Reg 00 BC 01 DE 10 IY 11 SP
INC ss	ss ← ss + 1	•	•	X	•	X	•	•	•	00 ss0 011		1	1	6	
INC IX	IX ← IX + 1	•	•	X	•	X	•	•	•	11 011 101 00 100 011	DD 23	2	2	10	
INC IY	IY ← IY + 1	•	•	X	•	X	•	•	•	11 111 101 00 100 011	FD 23	2	2	10	
DEC ss	ss ← ss − 1	•	•	X	•	X	•	•	•	00 ss1 011		1	1	6	
DEC IX	IX ← IX − 1	•	•	X	•	X	•	•	•	11 011 101 00 101 011	DD 2B	2	2	10	
DEC IY	IY ← IY − 1	•	•	X	•	X	•	•	•	11 111 101 00 101 011	FD 2B	2	2	10	

NOTES: ss is any of the register pairs BC, DE, HL, SP.
pp is any of the register pairs BC, DE, IX, SP.
rr is any of the register pairs BC, DE, IY, SP.

Flag Notation: • = flag not affected. 0 = flag reset. 1 = flag set. X = flag is unknown.
↕ = flag is affected according to the result of the operation.